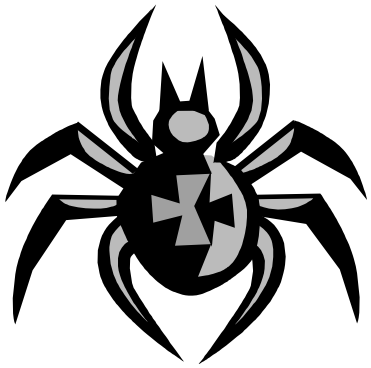
CAPÍTULO 1

El Modelo de Objetos

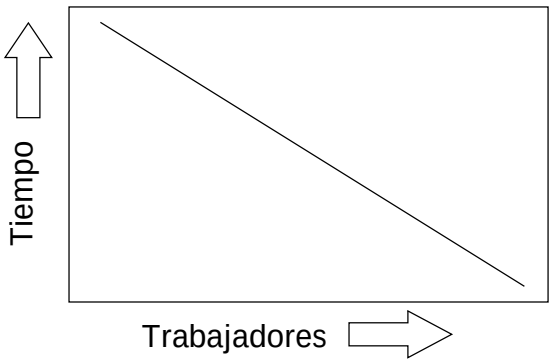
Desarrollo de Software a Gran Escala

¿Extrapolable?



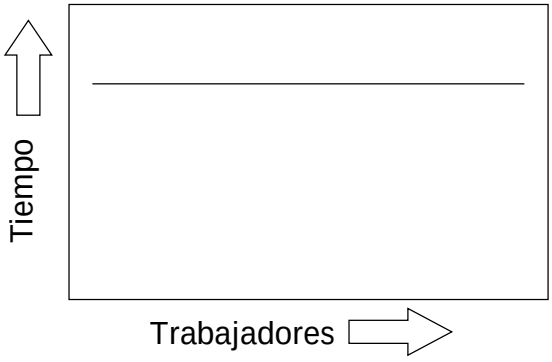
2

Tiempo vs. Trabajadores
Labor Perfectamente Particionable



3

Tiempo vs. Trabajadores
Labor No Particionable



4

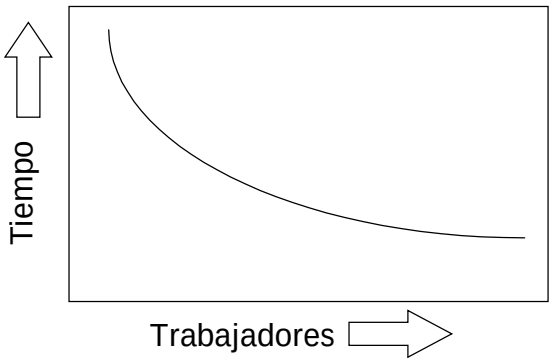
Principio de Brooks

Gestar un bebé toma nueve meses, sin importar cuantas mujeres se asignen a esta labor.

5

Tiempo vs. Trabajadores

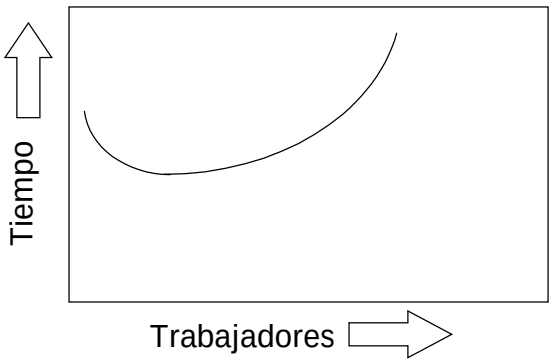
Labor Particionable con Comunicación



6

Tiempo vs. Trabajadores

Labor con Interrelaciones Complejas



7

Ley de Brooks

Agregar personal a un proyecto de software que está retrasado lo retrasa aún más.

8

Tendencias en la Ingeniería de Software

- Cambio de enfoque de la programación a pequeña escala hacia la programación a gran escala.
- Evolución de lenguajes de programación de alto nivel.

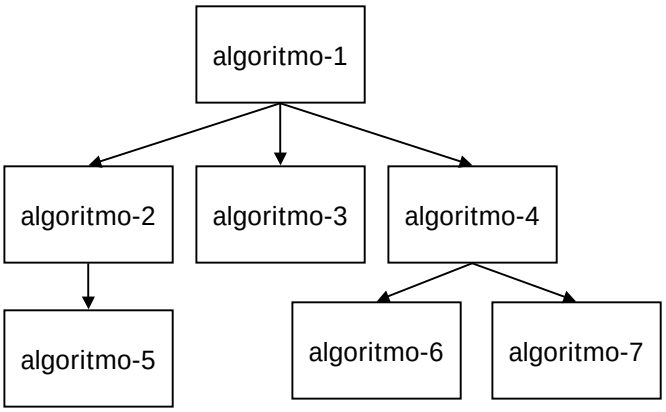
9

Descomposición: "*divide et impera*"

- Para dominar la complejidad de un sistema, se debe dividir éste en partes más pequeñas.
- En el desarrollo de software existen dos enfoques para descomponer un sistema:
  - Descomposición algorítmica
  - Descomposición orientada a objetos

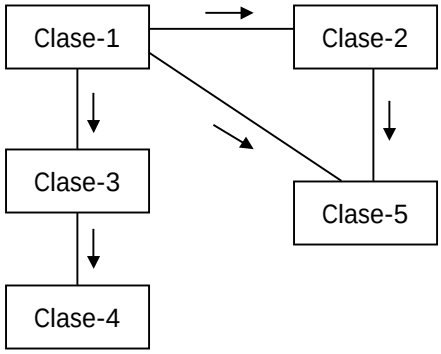
10

Descomposición Algorítmica



11

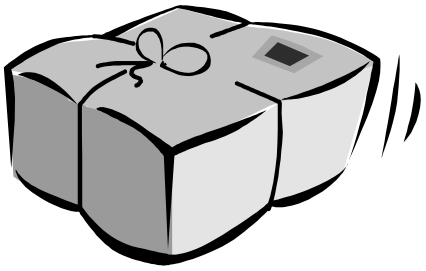
Descomposición Orientada a Objetos



12

Objeto

- Un objeto es un entidad que combina propiedades de procedimientos y datos *(realiza operaciones de cómputo y mantiene un estado local).*



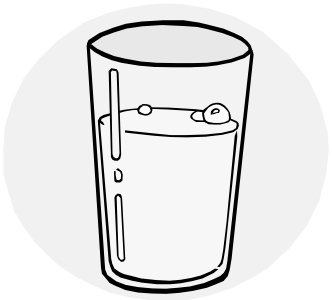
13

- Un objeto consta de:
  - ❑ Estado (atributos)
  - ❑ Comportamiento (operaciones)
  - ❑ Identidad
- La estructura y el comportamiento de objetos similares se definen en una *clase* común.
- A un objeto también se le llama *instancia de una clase* o simplemente *instancia*.

14

Ejemplo de Objeto

- El objeto vaso tiene el siguiente estado: *cantidad de líquido*.
- Su comportamiento es: *llenar*, *vaciar*, *arrojar*.
- Identidad: *Dos vasos del mismo estilo son similares pero distintos entre sí.*



15

Clase

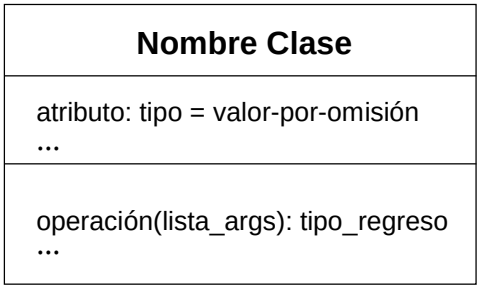
- A un conjunto de objetos que comparten una estructura y comportamiento común se le llama *clase*.
- Comúnmente, los términos clase y tipo se usan de manera intercambiable.

16

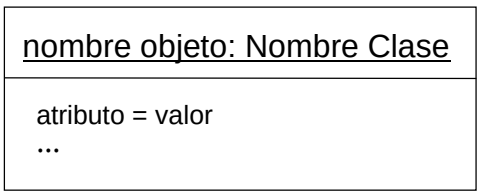
UML: Unified Modeling Language

- El *Lenguaje de Modelado Unificado* es una notación de propósito general para especificar y visualizar proyectos de software orientados a objetos.
- Desarrollado por los “three amigos”: Grady Booch, James Rumbaugh e Ivar Jacobson.
- Aprobado por el Object Management Group (OMG) en 1997.

17



Notación UML para clases



Notación UML para instancias

18

Mensajes y Métodos

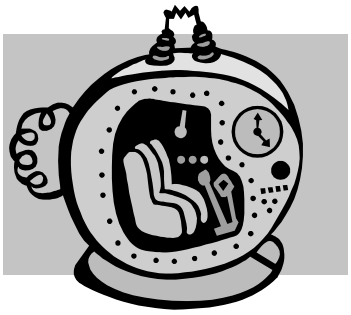
- Para solicitar que un objeto lleve a cabo alguna acción, se le envía un *mensaje*.
- Al recibir el mensaje, el objeto ejecuta un *método*, el cual es definido por la clase a la que pertenece éste.

19

- Por tanto, un mensaje es el qué de la acción, mientras que un método es el cómo.
- Objetos de distintas clases pueden responder al mismo mensaje de manera diferente.
- Esto significa que todo mensaje debe tener asociado al menos un método.

20

Encapsulación



- Al proceso de colocar en un compartimento los elementos de una abstracción que constituyen su estructura y comportamiento se le llama *encapsulación*.

21

- La encapsulación se logra a través del *ocultamiento de información*, que consiste en esconder todos los secretos de un objeto que no contribuyen a las características esenciales de éste.
- La encapsulación sirve para separar la *interfaz* (parte pública) de una abstracción de su *implementación* (parte privada).

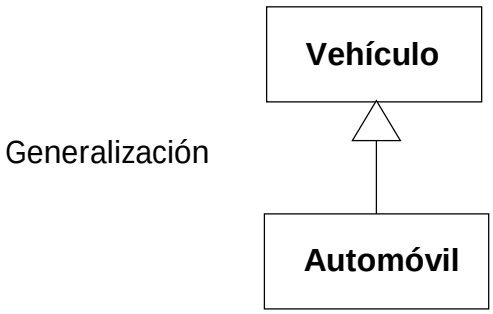
22

Relaciones entre Clases

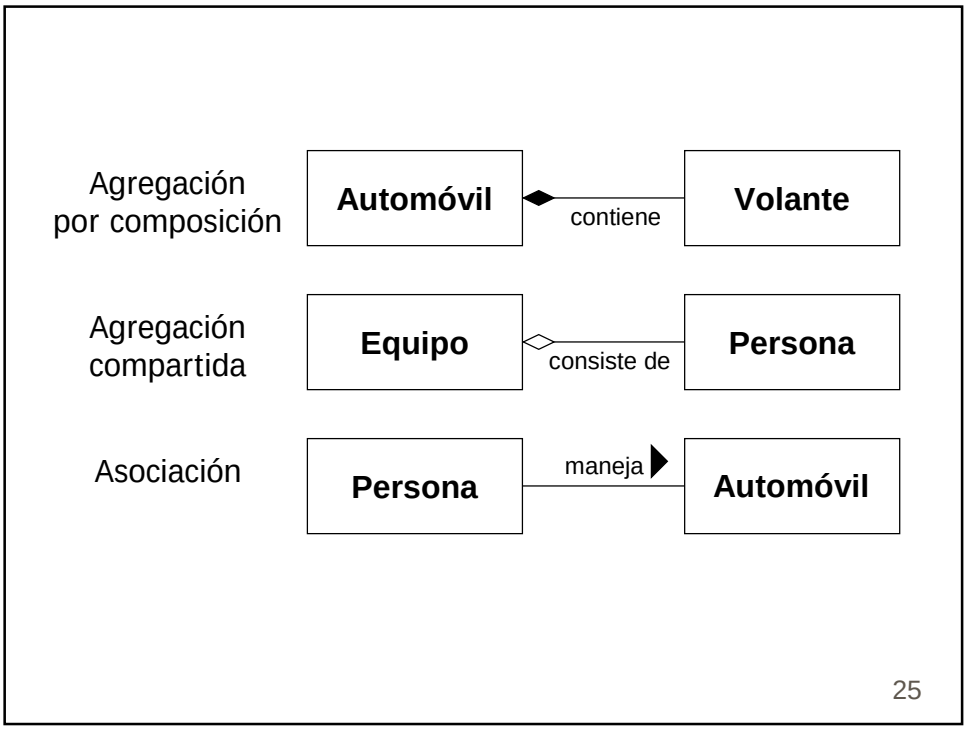
- GENERALIZACIÓN (HERENCIA):
  *Relación de tipo **es-un**.*
- AGREGACIÓN (PERTENENCIA):
  *Relación de tipo **tiene-un**.*
- ASOCIACIÓN (USO):
  *Relación de tipo **usa-un**.*

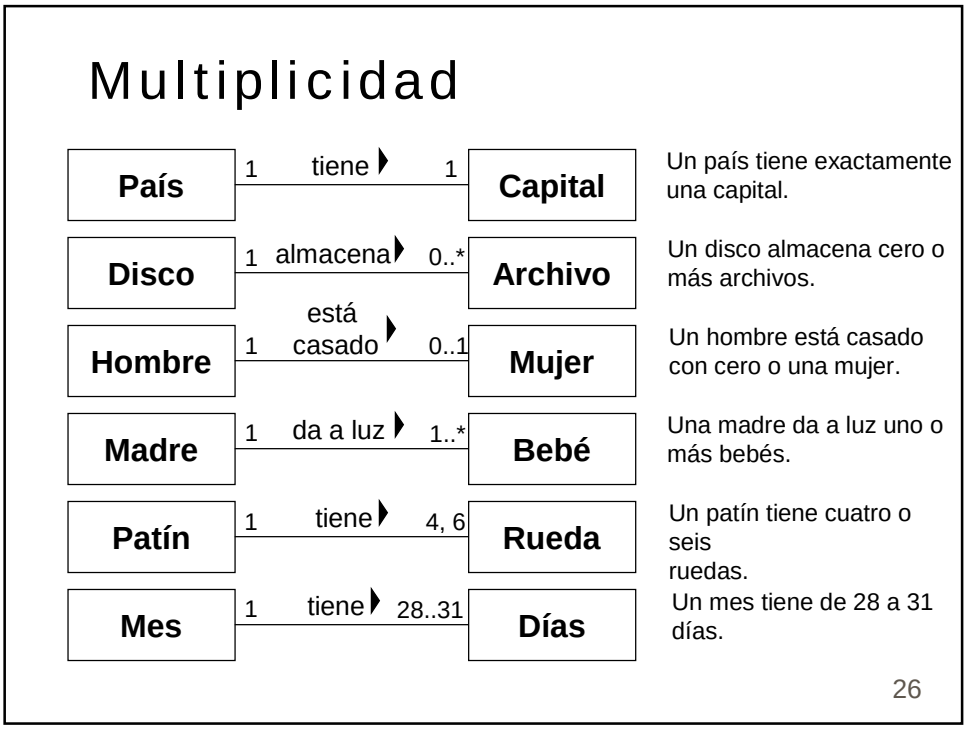
23

Ejemplos de Relaciones entre Clases y su Notación



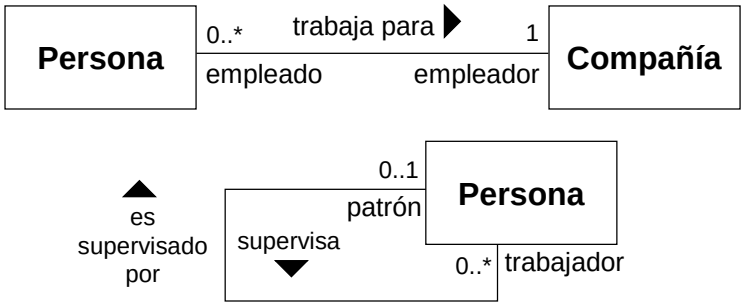
24





Nombre de Rol

- Un *rol* es la punta de una relación de asociación. Un *nombre de rol* es un nombre que identifica de manera única a cada punta de la asociación.



27

Herencia

- Una clase comparte la estructura y comportamiento definidos en una (herencia simple) o más (herencia múltiple) clases.

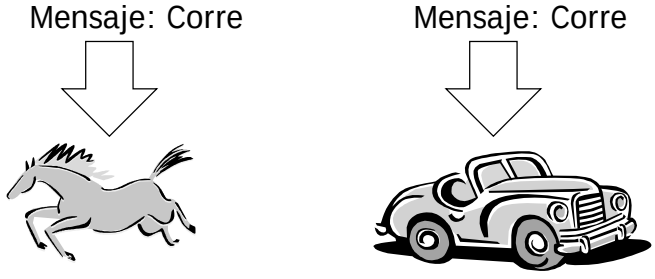
28

- La clase que recibe la herencia se le llama *sub-clase* o *clase derivada*.
- La clase de la cual se hereda se le llama *super-clase* o *clase base*.
- La herencia sirve para definir una jerarquía de tipo *es-un* entre clases en donde una clase derivada hereda de una o más clases base generalizadas.
- Típicamente, una clase derivada especializa su(s) clase(s) base incrementando o sobreponiendo el comportamiento y estructura existente.

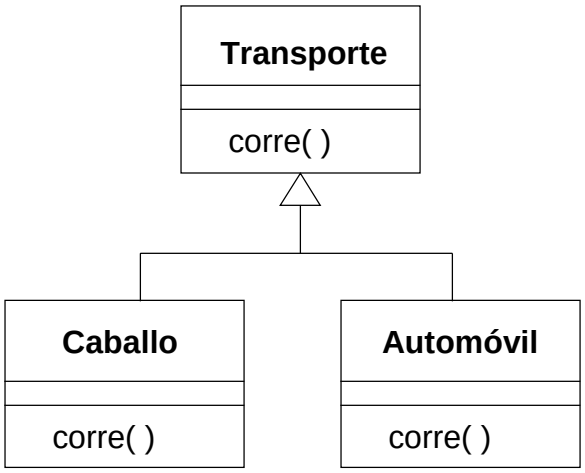
29

Polimorfismo

- *Polimorfismo* es cuando objetos de clases distintas, pero con una superclase común, pueden responder de manera diferente a un mismo mensaje.



30



31

- Sea *x* una variable que contiene un *Transporte*, a la cual se le manda el siguiente mensaje:

  x.corre( )

- La forma en que *x* responde al mensaje *corre* depende del objeto actualmente referido por *x*, que puede ser una instancia de la clase *Caballo* o *Automóvil*.

32

- Dado que el objeto denotado por *x* solamente puede determinar cuando el programa se está ejecutando, el polimorfismo sólo se puede llevar a cabo a tiempo de corrida.

33

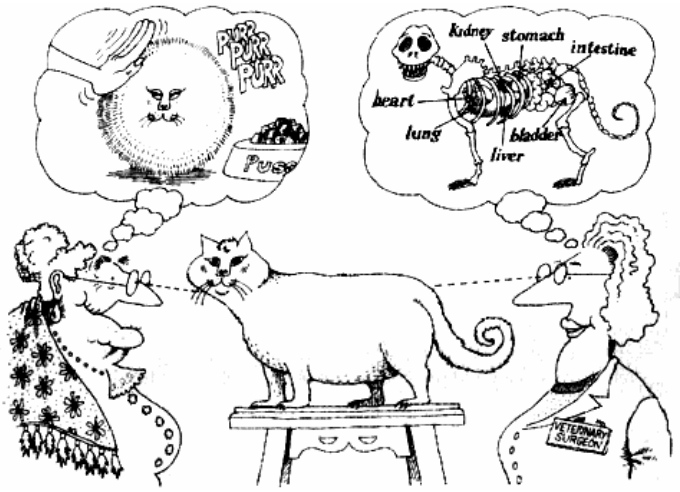
Elementos del Modelo de Objetos

- Cada paradigma de programación está basado en un marco conceptual que requiere de un razonamiento particular. En cuanto la orientación a objetos, el marco conceptual es el *modelo de objetos*.
- A continuación se presenta el modelo de objetos propuesto por Grady Booch.

34

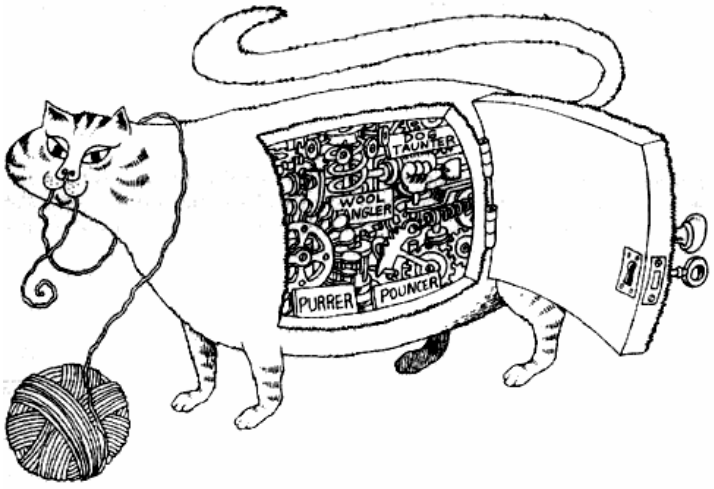
- Elementos Mayores (indispensables):
  - Abstracción
  - Encapsulación
  - Modularidad
  - Jerarquía
- Elementos Menores (opcionales):
  - Manejo estricto de tipos
  - Concurrencia
  - Persistencia

35



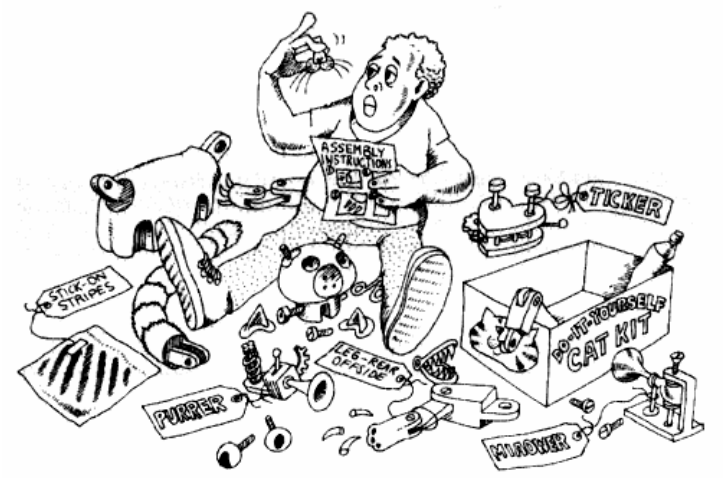
La **abstracción** se enfoca en las características esenciales de algún objeto, relativo a la perspectiva del espectador.

36



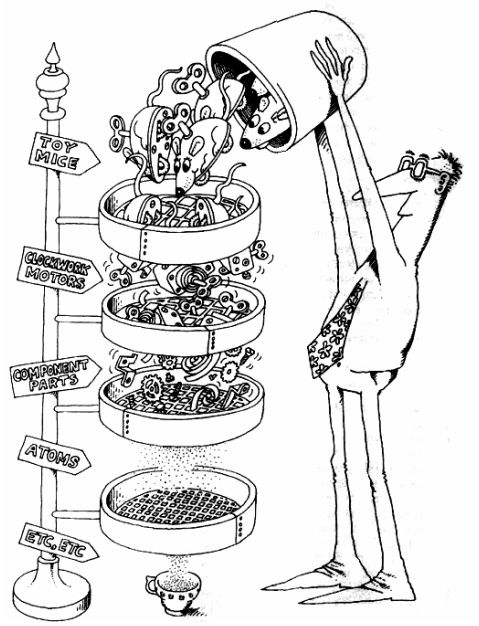
La **encapsulación** esconde los detalles de la implementación de un objeto.

37



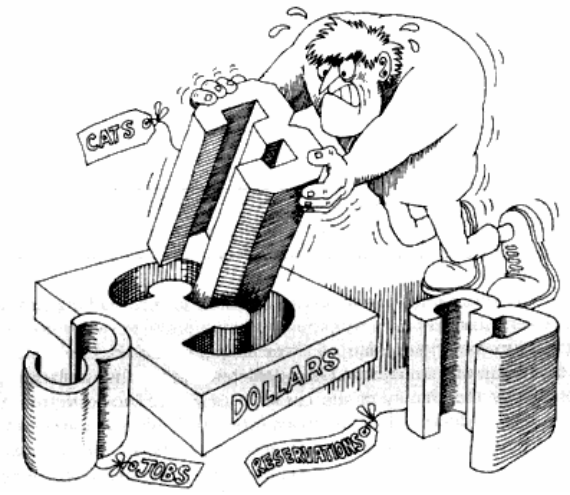
La **modularidad** empaqueta las abstracciones en unidades discretas.

38



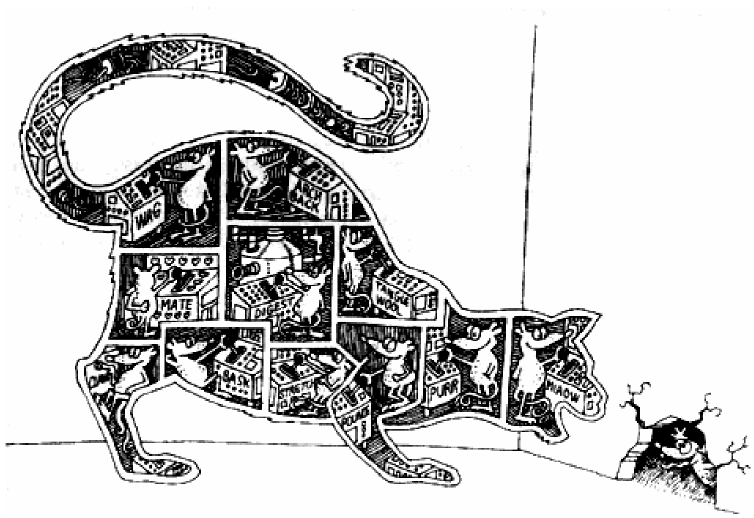
Las abstracciones conforman una **jerarquía**.

39



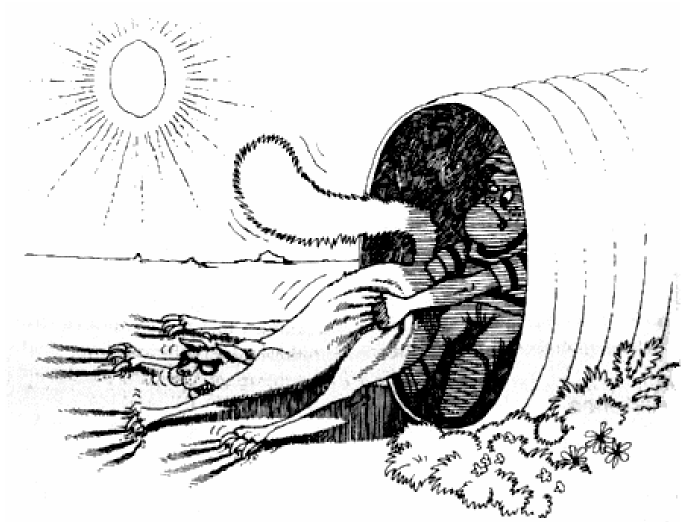
El **manejo estricto de tipos** previene mezclar indebidamente abstracción.

40



La **concurrencia** permite a distintos objetos actuar al mismo tiempo.

41



La **presistencia** almacena el estado de un objeto a través del tiempo y/o espacio.

42

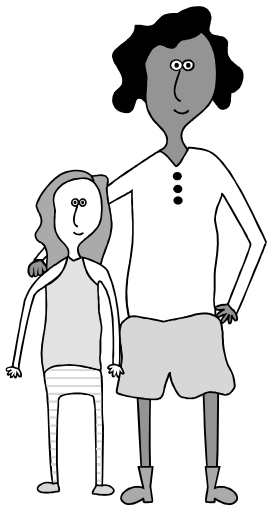
CAPÍTULO 2

Diseño Orientado a Objetos

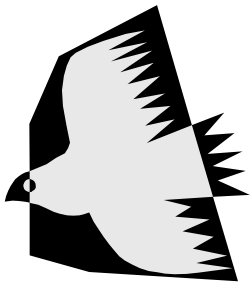
El aspecto más importante de la Construcción de Software Orientado a Objetos es la técnica de diseño que consta en delegar responsabilidades.

44

Cuando se hace que un objeto sea responsable, se espera un determinado comportamiento, por lo menos cuando se observan ciertas reglas.



45

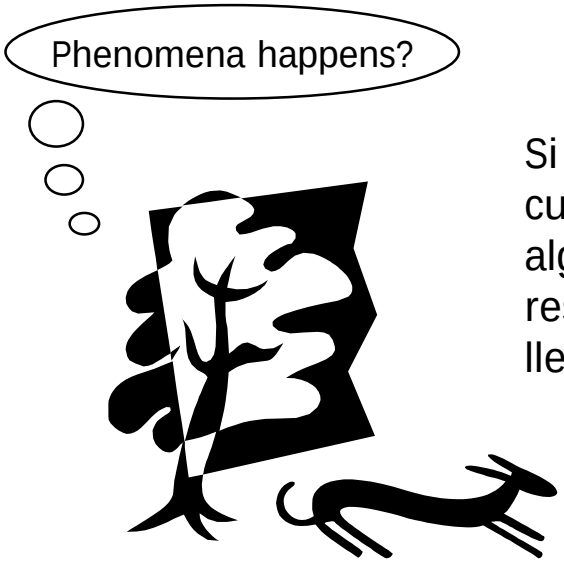


La responsabilidad implica un cierto grado de independencia o no interferencia.

46

Cuando se diseñan aplicaciones orientadas a objetos, es útil pensar que este proceso es similar a organizar a un grupo de individuos.

47



Si debe ocurrir cualquier acción, alguien debe ser responsable de llevarla a cabo.

48

El buen diseño orientado a objetos consiste en establecer quién es responsable de cada acción que se debe llevar a cabo.

49

Tarjetas CRC

- Una tarjeta CRC es una tarjeta de cartón de 4 x 6 pulgadas que sirve para determinar las responsabilidades y los colaboradores de una clase.

50

Estructura de una tarjeta CRC

Nombre de la Clase	Colaboradores
Responsabilidades	

51

- Cada clase de un diseño se describe en una tarjeta por separado. La distinción entre instancias y clases se nubla al momento de realizar el diseño.
- Debajo del nombre de la clase, se puede indicar con qué superclases y subclases se tiene relación.
- En la parte opuesta de la tarjeta se puede colocar la siguiente información:
  - Breve descripción de la clase.
  - Atributos (datos) de la clase.

52

Escenarios

- Cuando se diseña utilizando tarjetas CRC no se siguen los modelos tradicionales desarrollo “descendente” o “ascendente”.

53

- El diseño progresa de lo desconocido a lo conocido.
- Se comienza con las clases más obvias que se requieren para iniciar la aplicación.
- Se comienza a jugar *“que ocurriría si”*, simulando los escenarios que ilustran el uso esperado.
- En un equipo de diseño, cada persona es responsable de una o más clases.

54

- Cuando una clase se hace relevante en el escenario, su tarjeta es sostenida mientras se “ejecuta”.
- El flujo de programa se percibe como el movimiento que ocurre de una tarjeta hacia otra.
- Al momento en que se identifica una acción en el escenario, se le asigna una responsabilidad a una clase.
- La responsabilidad es anotada en la tarjeta correspondiente.

55

- Se deben intentar diferentes escenarios, incluyendo condiciones de error y situaciones excepcionales, generando más responsabilidades.
- La tarea se convierte en desarrollar un entendimiento y una clara descripción de cada componente del sistema.
- Durante el proceso, probablemente se detecte la necesidad de crear nuevas clases, o de mover ciertas responsabilidades de una clase hacia otra.

56

Ejemplo: Un Cajero Automático

57

LectorTarjeta	Colaboradores
- Desplegar bienvenida. *- Tomar tarjeta.* *- Pedir al verificador de NIP que realice validación.* *- Llamar al selector de actividad.* *- Regresar tarjeta al usuario.*	VerificadorNIP SelectorActividad

58

VerificadorNIP	Colaboradores
- Tomar NIP del administrador de cuenta. *- Solicitar NIP del usario.* *- Comparar los NIPs; devolver resultado.*	AdminCuenta

59

AdminCuenta	Colaboradores
- Validar existencia de cuenta. *- Devolver NIP de cuenta.* *- Validar y realizar un retiro.*	

60

SelectorActividad	Colaboradores
- Desplegar menú de actividades. *- Esperar por selección del usuario.* *- Llamar al administrador de transacción correspondiente.*	AdminRetiro AdminDepósito

61

AdminRetiro	Colaboradores
- Solicitar al usuario la cantidad a retirar. *- Solicitar al administrador de cuenta llevar a cabo la transacción.* *- Entregar al usuario el efectivo solicitado.*	AdminCuenta

62

AdminDepósito	Colaboradores
- Solicitar al usuario la cantidad a depositar. *- Solicitar sobre con el depósito y emitir sello de hora y fecha.*	

63

Principio de la Administración de Datos

- Cualquier valor que deba ser leído y/o modificado ampliamente, o que exista por un periodo prolongado de tiempo, debe ser ***administrado***.

64

- Una y solamente una clase debe ser responsable de las acciones que se realicen para ver o alterar dicho valor. Todas las demás clases que requieran obtener estos valores deben solicitarlo explícitamente al administrador de dichas acciones, en lugar de intentar acceder directamente a los datos.

65

Utilización de Nombres

- Los objetos deben ser nombrados utilizando sustantivos propios, por ejemplo “elSensor” o simplemente “figura”.
- Las clases deben ser nombradas con sustantivos comunes, por ejemplo “Sensor” o “Figura”.

66

- Aquellas operaciones que produzcan modificaciones deben ser nombradas utilizando verbos activos, por ejemplo “dibuja” o “mueveIzquierda”.
- Los nombres que representan valores booleanos deben permitir una fácil interpretación. Por ejemplo “impresoraLista” en lugar de “estadoImpresora”.

67

- Examinar cuidadosamente las abreviaciones. Ciertas abreviaciones pueden significar diferentes cosas para distintas personas, por ejemplo “procTerm”.
- Utilizar nombres que sean pronunciables. Si no se puede leer en voz alta un nombre, no es un buen nombre.

68

- Utilizar letras mayúsculas o el carácter de subrayado para marcar el inicio de una nueva palabra dentro de un nombre, tal como “VerificaIdentidad” o “Verifica_identidad”, en lugar de “Verificaidentidad”.

69

Descubriendo Herencia

- Al utilizar tarjetas CRC, se pueden establecer relaciones de herencia si esto resulta obvio.
- En general, la herencia se *debe empezar a buscar una* vez que se haya establecido un diseño inicial.

70

- Para descubrir una relación de herencia se deben encontrar comportamientos similares entre dos o más clases.
- No se debe perder de vista que la relación de herencia es una relación de tipo ***es-un***.
- Es posible también que las clases propuestas en el diseño puedan ser derivadas a partir de clases previamente definidas (en otros proyectos o en bibliotecas de clases).

71

Errores Comunes de Diseño

- Clases que modifican directamente a otras clases.
- Clases con demasiadas responsabilidades.

72

- Clases sin responsabilidad alguna.
- Clases con responsabilidades inútiles.
- Nombres engañosos.
- Responsabilidades no relacionadas.
- Uso inadecuado de herencia.
- Funcionalidad repetida.

73

CAPÍTULO 3

La Plataforma Java

En el inicio fue Oak...

- Desarrollado por James Gosling en Sun Microsystems, Mountain View, California.
- Surge como lenguaje para programar aplicaciones incrustadas en aparatos electrónicos de consumo.

75

- Comenzó como una reimplementación de C++.
- Siempre se consideró una herramienta, nunca un fin en sí mismo.
- Al surgir el Web ocurrió un matrimonio inesperado pero bien recibido …
- y entonces nació Java.

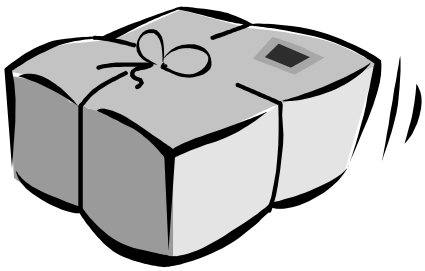
76

Java™

- Orientado a objetos
- Sencillo
- Tipos estrictos
- Administración automática de memoria
- Concurrente
- Distribuido
- Dinámico
- Compatibilidad binaria
- Seguro

77

Orientado a Objetos



- Las clases son el único medio para definir el código y los datos de un programa.
- Los tipos primitivos (int, char, float) no se manejan como objetos.

78

Sencillo

- Sintaxis familiar para los programadores de C y C++.
- No tiene apuntadores.
- Cuenta con una biblioteca muy completa de clases listas para usarse.

79

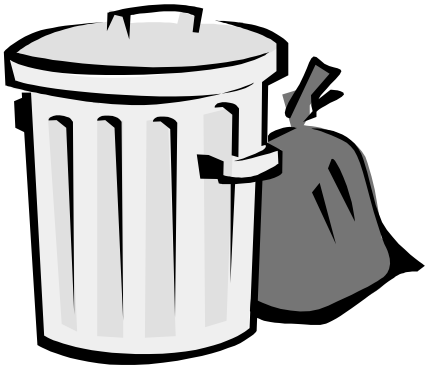
Tipos Estrictos

- Todos los objetos en un programa son de un determinado tipo.
- El compilador de verifica que los objetos se utilicen en contextos que sean compatibles con su tipo (no se pueden sumar peras con manzanas).

80

Administración Automática de Memoria

- Cuando se crea un objeto, éste es ubicado en el montículo (*heap*) del sistema.
- Cuando el objeto ya no es requerido (no existe forma de llega a él), el sistema lo reclama de manera automática mediante un esquema de *recolección de basura*.



81

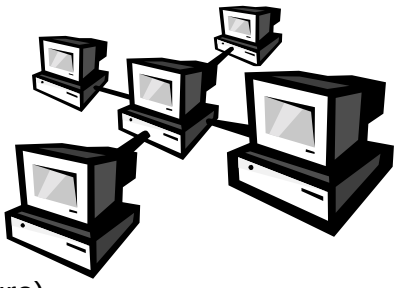
Concurrente

- Soporte de *threads* (procesos ligeros).
- Un *thread* es una parte de un programa que se ejecuta al mismo tiempo que otras partes.

82

Distribuido

- Incorpora un soporte de alto nivel para las redes de computadora:
  - URLs
  - TCP (Sockets)
  - UDP (Datagramas)
  - RMI (Remote Method Invocation)
  - CORBA (Common Object Request Broker Architecture)
  - Servlets



83

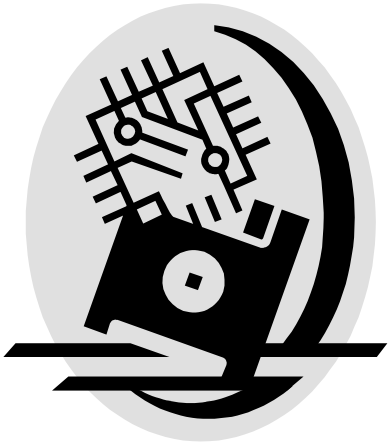
Dinámico

- Cualquier clase se puede cargar y ligar a tiempo de ejecución.
- Se puede obtener toda la información de una clase a tiempo de corrida mediante el mecanismo de *reflexión*.

84

Compatibilidad Binaria

- Los programas se compilan a un formato de *byte-code* de arquitectura neutral.
- Cada plataforma específica requiere implementar una máquina virtual que ejecute los *byte-codes*.



85

Seguro

- Incorpora diversos esquemas que colaboran en la seguridad de los usuarios, desarrolladores y administradores:
  - Permisos
  - Criptografía
  - Certificación

86

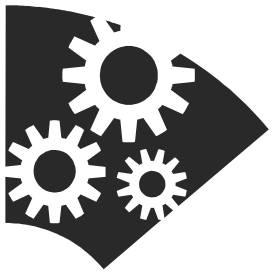
Java en la actualidad

- Amplia adopción, convirtiéndose rápidamente en un ambiente universal.
- Beneficios para desarrolladores, usuarios y administradores a través de intranets e Internet.

87

- Varias compañías cuentan ya con sofisticadas herramientas de desarrollo: JavaSoft, Symantec, Inprise, Microsoft, IBM, Oracle, etc.
- Aplicaciones, sistemas, dispositivos y chips.

88

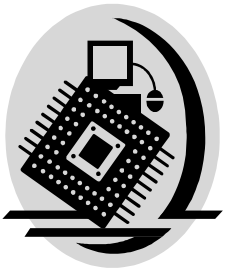


La Máquina Virtual de Java

- La JVM (*Java Virtual Machine*) es Clave en la independencia de plataforma.
- Tecnología implementada en muchas plataformas: Windows 95/98/NT, MacOS, OS/2, OS/400, Linux, Solaris, HP-UX, AIX, etc.

89

- La JVM puede estar implementada como:



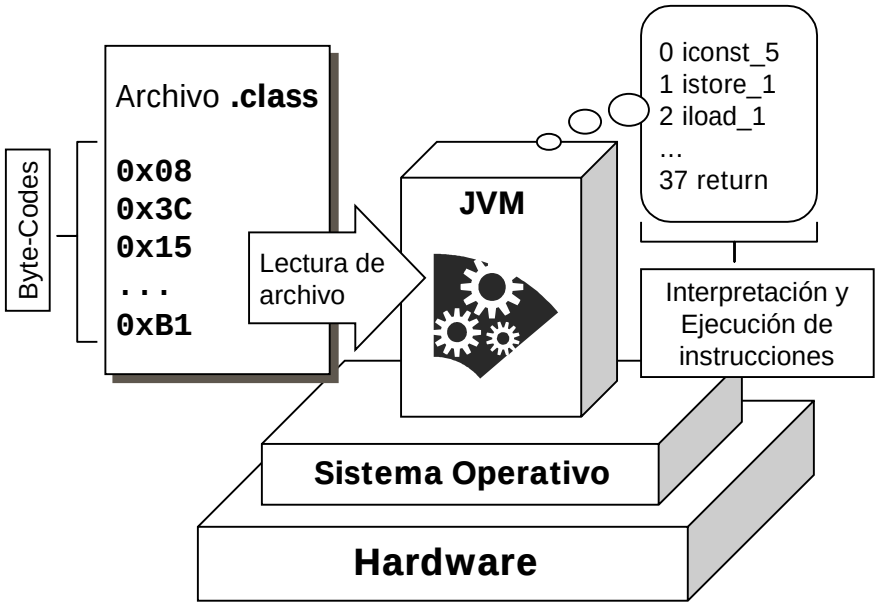
- una aplicación de software “stand-alone” independiente (por ejemplo: JRE).
- parte integral de un visualizador (*browser*) de Web (por ejemplo: HotJava).
- un componente integral del sistema operativo (por ejemplo: JavaOS).
- un núcleo de microprocesador o chip controlador (por ejemplo: picoJava) para productos electrónicos de consumo (tarjetas inteligentes, telefonos celulares, agendas electrónicas, etc.).

90

Archivo de Clase

- Un compilador de Java compila un programa fuente y produce un archivo de clase (**.class**) que contiene una secuencia de *byte-codes* (códigos de bytes).
- La JVM lee el archivo de clase e interpreta y ejecuta las instrucciones ahí contenidas.

91



92

- Ventajas del archivo de clase:
  - Transportable entre diferentes plataformas de cómputo.
  - Archivos binarios pequeños, muy conveniente para transferirse a través de redes.
  - El contenido es verificable, con lo que se evita la ejecución de archivos corruptos.
  - El formato de *byte-code* puede ser generado por compiladores y otras herramientas distintas a Java.
  - Permite que se oculte el código fuente, a diferencia de los programas escritos como scripts (ejemplo: JavaScript y VBScript).

93

CAPÍTULO 4

Elementos Básicos

Hola Mundo

- El programa Hola Mundo en Java:

```
// Programa Hola Mundo
class HolaMundo {
    public static void main(String[] args) {
        System.out.println("¡Hola mundo!");
    }
}
```

95

Comentarios

- Java soporta tres tipos de comentarios:
  - Comentario de bloque
    `/* texto */`
  - Comentario de línea
    `// texto`
  - Comentario de documentación
    `/** documentación */`

96

Declaración de Variables Locales

- Toda variable local debe ser declarada antes de ser usada.
- Dichas declaraciones pueden ocurrir en cualquier parte del cuerpo de un método.
- Una declaración consiste de un tipo, el nombre de la variable, y opcionalmente un inicializador.

97

- Ejemplos:

```
// declara una variable
int unEntero;

/* declara una variable y le da un valor
   inicial */
int otroEntero = 10;
```

98

Identificadores

- Un identificador de Java sirve para nombrar entidades declaradas.
- Deben comenzar con una letra (incluyendo los caracteres _ y **$**) y continuar con cero o más letras o dígitos.
- Los identificadores son sensibles a mayúsculas y minúsculas.
- Los identificadores pueden ser de cualquier longitud.

99

- Las siguientes palabras son reservadas y no pueden ser utilizadas como identificadores:

```
abstract  double      int           super
boolean   else        interface     switch
break     extends     long          synchronized
byte      final       native        this
case      finally     new           throw
catch     float       package       throws
char      for         private       transient
class     goto *      protected     try
const *   if          public        void
continue  implements  return        volatile
default   import      short         while
do        instanceof  static
```

* no se usan actualmente

100

- Las literales **`null`**, **`true`** y **`false`** no son técnicamente palabras reservadas, sin embargo tampoco pueden ser utilizadas como nombres de identificadores.

101

Unicode

- Las letras y dígitos reconocidos en Java son los definidos por el conjunto de caracteres de Unicode.
- Los códigos ASCII (7 bits) e ISO Latin-1 (8 bits) son subconjuntos de Unicode (16 bits).
- Un archivo fuente de Java en ASCII o Latin-1 se traduce primero a Unicode antes de ser procesado.

102

Tipo de Datos Primitivos

Tipo	*Valores Posibles*
boolean	true, false
char	Unicode 16 bits
byte	Entero con signo de 8 bits complemento a 2
short	Entero con signo de 16 bits complemento a 2
int	Entero con signo de 32 bits complemento a 2
long	Entero con signo de 64 bits complemento a 2
float	Número de punto flotante de 32 bits IEEE 754
double	Número de punto flotante de 64 bits IEEE 754

103

Literales

- Una literal es la forma en que se escribe una constante de un cierto tipo.
- Las literales de tipo booleano son **`true`** y **`false`**.
- Las literales enteras pueden ser representadas en base decimal, octal y hexadecimal:

```
29 035 0x1D
```

104

- Si la literal termina con **`L`** o **`l`**, entonces es de tipo **`long`**:

  `29L`
- Las literales de punto flotante pueden ser de tipo **`double`** si no tiene sufijo alguno o si tienen el sufijo **`D`** o **`d`**:

  `18.  .2  0.0  0d  1e5`
- Las literales de punto flotante pueden ser de tipo **`float`** si tienen el sufijo **`F`** o **`f`**:

  `18.f  .2F  0.0F  0f  1e5f`

105

- Las literales de carácter aparecen entre comillas sencillas: **`'A'`**
- Las literales que representan cadenas de caracteres se indican entre comillas dobles:

  `"una cadena"`
- Las cadenas de caracteres pueden contener secuencias de escape para indicar ciertos caracteres especiales:

  `"\tuno\tdos\ttres\n"`

106

Secuencias de Escape

`\u`*`NNNN`*	Carácter Unicode hexadecimal	
`\b`	BS: Retroceso	(`\u0008`)
`\t`	HT: Tabulador	(`\u0009`)
`\n`	LF: Nueva línea	(`\u000A`)
`\f`	FF: Nueva página	(`\u000C`)
`\r`	CR: Retorno de carro	(`\u000D`)
`\"`	Comillas dobles	(`\u0022`)
`\'`	Comilla sencilla	(`\u0027`)
`\\`	Diagonal inversa	(`\u005C`)
`\`*`NNN`*	Carácter en valor octal	

107

Operadores Aritméticos

+ op	Más unario
- op	Menos unario
op1 * op2	Multiplicación
op1 / op2	División
op1 % op2	Residuo
op1 + op2	Suma
op1 - op2	Resta

108

Operadores de Incremento y Decremento

`++ op`	Preincremento
`op ++`	Postincremento
`-- op`	Predecremento
`op --`	Postdecremento

109

Operadores Para Manejo de Bits

`~ op`	Complemento a uno
`op1 & op2`	AND de bits
`op1 | op2`	OR de bits
`op1 ^ op2`	XOR de bits
`op1 << op2`	corrimiento a la izquierda (insertar ceros por la derecha)
`op1 >> op2`	corrimiento a la derecha (insertar el bit más significativo por la izquierda)
`op1 >>> op2`	corrimiento a la derecha (insertar ceros por la izquierda)

110

Operadores Relacionales y de Igualdad

op1 > op2	Mayor que
op1 >= op2	Mayor o igual a
op1 < op2	Menor que
op1 <= op2	Menor o igual a
op1 == op2	Igual a
op1 != op2	Diferente de

111

Operadores Lógicos

! op1	NOT lógico
op1 & op2	AND lógico (evaluación estricta)
op1 | op2	OR lógico (evaluación estricta)
op1 ^ op2	XOR lógico (evaluación estricta)
op1 && op2	AND lógico (evaluación de corto circuito)
op1 || op2	OR lógico (evaluación de corto circuito)

112

Operadores de Asignación

`op1 = op2`	Asignación simple
`op1 += op2`	Asignación compuesta con suma
`op1 -= op2`	Asignación compuesta con resta
`op1 *= op2`	Asignación compuesta con multiplicación
`op1 /= op2`	Asignación compuesta con división
`op1 %= op2`	Asignación compuesta con residuo
`op1 <<= op2`	Asignación compuesta con corrimiento izquierdo
`op1 >>= op2`	Asignación compuesta con corrimiento derecho (preservando signo)
`op1 >>>= op2`	Asignación compuesta con corrimiento derecho (sin preservar signo)
`op1 &= op2`	Asignación compuesta con AND
`op1 |= op2`	Asignación compuesta con OR
`op1 ^= op2`	Asignación compuesta con XOR

113

Conversión entre Tipos de Datos

- Las conversiones entre tipos de datos distintos pueden ser de dos formas:
  1. Ensanchamiento (*widening*)
  2. Estrechamiento (*narrowing*)

114

Conversión por Ensanchamiento

- Ocurre cuando un tipo de dato se convierte a otro tipo de mayor precisión. Por ejemplo de **`byte`** a **`int`**, de **`char`** a **`int`**, de **`float`** a **`double`**.
- Este tipo de conversiones se hacen de manera implícita y no involucran perdida de información ni precisión. Por ejemplo:

```
byte b = 100;          float f = 1.2;
int i = b;             double d = f;
```

115

Conversión por Estrechamiento

- Ocurre cuando se desea convertir de un tipo de mayor precisión a uno de menor.
- La conversión por estrechamiento se lleva a cabo a través de una conversión explícita o *cast*. Dicha conversión puede involucrar una pérdida de precisión o información.

```
short s = 511;
byte b = (byte) s;
double d = 7.99;
long l = (long) d;
```

116

Operador Condicional **?:**

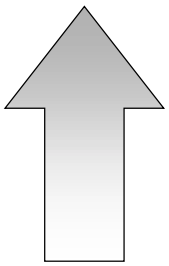
condición **?** *consecuente* **:** *alternativa*

```
int x, y = 0, z = 5;
x = y == z - z ? z * 2 : y - 1;
```

117

```
() op++ op--
++op --op +op -op ~ !
(tipo)op
* / %
+ -
<< >> >>>
< <= > >=
== !=
&
^
|
&&
||
?:
= *= /= %= += -= <<= >>= >>>= &= ^= |=
```

Mayor precedencia



Precedencia de operadores

118

Asociatividad de Operadores

- Todos los operadores binarios son asociativos por la izquierda, a excepción de los operadores de asignación los cuales se asocian por la derecha.

```
a + b + c          a = b = c
(a + b) + c        a = (b = c)
```

119

Sentencia de Bloque

- Las llaves (**{** y **}**) agrupan cero o más sentencias en un bloque.
- Las variables locales existen solamente mientras se ejecuta el bloque en el que fueron declaradas.
- Las variables locales deben ser inicializadas antes de usarse, de otra forma se produce un error de compilación.

120

Sentencia **if**

```
if (expresión-booleana)
    sentencia1

if (expresión-booleana)
    sentencia1
else
    sentencia2
```

121

Sentencia **switch**

```
switch (expresión-entera) {
case cte1:
    sentencia1
case cte2:
    sentencia2
...
default:
    sentenciaN
}
```

122

Sentencia **while** y **do-while**

while (*expresión-booleana***)**
sentencia

do
sentencia
while (*expresión-booleana***);**

123

Sentencia **for**

for (*expr-inic***;** *epr-bool***;** *expr-incr***)**
sentencia

- expr-inic y expr-incr pueden ser una lista expresiones separadas por comas.
- expr-inic puede incluir declaraciones de variables cuyo ámbito es exclusivamente la estructura del **for**.

124

Etiquetas

- Una sentencia puede estar etiquetada. Las etiquetas se utilizan generalmente en bloques y ciclos.
- Una etiquete precede a una sentencia de la siguiente forma:

  etiqueta`:` *sentencia*
- Un sentencia etiquetada es útil cuando se usa **`break`** y **`continue`**.

125

Sentencia **`break`**

- La sentencia **`break`** sirve para terminar un ciclo o una sentencia **`switch`**.
- El **`break`** puede ir seguido de una etiqueta, en cuyo caso termina la sentencia etiquetada correspondiente.

126

Sentencia **continue**

- La sentencia **continue** se salta hasta el final del cuerpo de un ciclo y evalúa la expresión booleana.
- El **continue** puede también contener una etiqueta , en cuyo caso el flujo del programa continúa hasta el final sentencia etiquetada, procediendo a evaluar la expresión booleana de ésta.

127

Ejemplo de uso de etiquetas

- Un ejemplo de uso de etiquetas junto con las sentencias **break** y **continue** se presenta a continuación:

```
int x = 0;
uno: for (int i=0;i<3;i++)
    dos: for (int j=0;j<3;j++)
        if (i + j > 1) {
            x ++;
            break dos;
        } else
            continue uno;
```

128

Arreglos

- Los arreglos son una colección de varios datos del mismo tipo. A cada componente de un arreglo se le puede acceder a través de un índice entero.
- La forma general de declarar una variable ***varArreglo*** de tipo arreglo, cuyos componentes son de tipo ***T***, es la siguiente:

```
T[] varArreglo;
```

129

- La declaración por sí sola del arreglo no lo crea. Se requiere utilizar el operador **new** para crear una nueva instancia de la clase arreglo de la siguiente forma:

```
varArreglo = new T[tamaño];
```

en donde ***tamaño*** es el número de elementos que se desea tener en el arreglo.

130

- Un arreglo creado con **new** automáticamente inicializa todos sus elementos a uno de los siguientes valores, dependiendo de su tipo:

Tipo	*Valor inicial*
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

131

- Otra forma de crear un arreglo es inicializándolo de manera explícita en su misma declaración:

  $T[]\ varArreglo = \{x_1, x_2, \ldots, x_n\}$

  donde $x_1, x_2, \ldots, x_n$ son los valores con los que se desea inicializar el arreglo, los cuales deben ser del tipo ***T***. El tamaño de dicho arreglo es ***n***.

132

- Los índices del arreglo son enteros en el siguiente rango:

  `0 <= `*`índice`*` <= `*`tamaño`*` - 1`

- Cada elemento del arreglo se accede con el nombre del arreglo seguido del operador **`[]`** colocando entre éstos el índice correspondiente.
- El tamaño de un arreglo está disponible en la variable de instancia **`length`**.

133

- Ejemplos de uso de arreglos:

```
int[] ai = new int[3];
double[] ad = {1.2, 2.4, 5.0};
boolean[] ab = new boolean[3];

for (int i = 0; i < ai.length; i ++) {
    ai[i] = (int) ad[i];
    ab[i] = ai[i] % 2 == 0;
}

int[] ai2 = ai;
ai2[0] ++;
```

134

CAPÍTULO 5

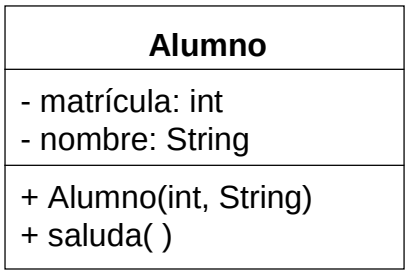
Clases

Clases

- Las clases definen la estructura, el comportamiento y los mecanismos de creación de las instancias de la clase.
- Las clases son parecidas a las estructuras (**`struct`**) de lenguajes como C, sin embargo, las clases de Java contienen también código (métodos) además de campos para almacenar datos.

136

Ejemplo: clase Alumno



137

Implementando la clase Alumno

```
class Alumno {
    private int matrícula;
    private String nombre;

    public Alumno(int matr, String nom) {
        matrícula = matr;
        nombre = nom;
    }
```

138

```
    public void saluda() {
        System.out.println("Hola, me llamo "
            + nombre + ", mi matrícula es: "
            + matrícula);
    }

    public static void main(String[] args) {
        Alumno a1 = new Alumno(430001, "Bety");
        Alumno a2 = new Alumno(431234, "Pepe");
        Alumno a3 = new Alumno(434321, "Paco");

        a1.saluda();
        a2.saluda();
        a3.saluda();
    }
} // class Alumno
```

139

Acceso a los Miembros de la Clase

- Para acceder a las variables de instancia y a los métodos de una clase se utiliza el operador punto (`.`) después de una variable que refiera a algún objeto:

  unObjeto.varInstancia

  ***unObjeto.método*(*args* …)**

140

- Todos los campos y métodos de una clase se pueden acceder desde cualquier parte de dicha clase.
- Para controlar el acceso por parte de otras clases, incluyendo subclases, los miembros de una clase cuentan con cuatro modificadores de control de acceso: *publico*, *privado*, *protegido* y *de paquete*.

141

Acceso público	Los miembros declarados `public` se pueden acceder en cualquier parte donde la misma clase sea accesible.
Acceso privado	Miembros declarados `private` son accesibles solamente dentro de la misma clase.
Acceso protegido	Miembros declarados `protected` son accesibles por la misma clase, sus subclases y por las clases contenidas en el mismo paquete.
Acceso de paquete	Miembros declarados sin modificador de acceso son accesibles por la misma clase y por las clases contenidas en el mismo paquete.

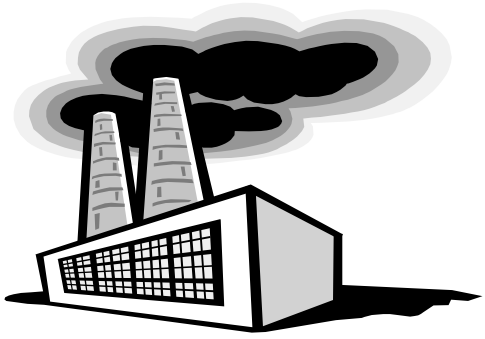
142

Accesos en UML

Nombre Clase
- Variable Privada + Variable Pública # Variable Protegida
- Método Privado + Método Público # Método Protegido

143

Creación de Objetos



- Para crear una nueva instancia de una clase se utiliza el operador **new** seguido del nombre de la clase y de una lista de argumentos entre paréntesis que determinan el constructor que se desea invocar.

144

- Dada la siguiente clase:

```
class Racional {
    public int numerador = 0;
    public int denominador = 1;
}
```

- Para crear una nueva instancia se puede usar el siguiente código:

```
// Crear la instancia Racional 1/2
Racional r = new Racional();
r.numerador = 1;
r.denominador = 2;
```

145

Constructores

- Si no se puede inicializar correctamente los campos de una instancia a través de una inicialización explícita, se debe recurrir a la definición de uno o más constructores.
- Los constructores tienen el mismo nombre que la clase pero, a diferencia de los métodos, no tienen tipo de regreso.

146

- Antes de que el constructor sea llamado, las variables de instancia son inicializadas con sus valores por omisión.
- Una clase puede tener varios constructores. Sin embargo se requiere que cada constructor tenga una *firma* única.
- El constructor que no recibe argumentos se le conoce como el constructor *no-arg*.

147

- Cada clase debe contar con al menos un constructor. Si la clase no cuenta con la declaración explícita de un constructor, el sistema proveerá un constructor *no-arg* por omisión.

148

- Modificando la clase Racional para restringir el acceso de las variables de instancia y añadir tres constructores:

```
class Racional {
    private int numerador;
    private int denominador;

    // Constructor no-arg.
    public Racional() {
        numerador = 0;
        denominador = 1;
    }
```

149

```
    // Constructor con un argumento
    public Racional(int num) {
        numerador = num;
        denominador = 1;
    }

    // Constructor con dos argumentos
    public Racional(int n, int d) {
        int mcd = máximoComúnDivisor(n, d);
        numerador = n / mcd;
        denominador = d / mcd;
    }
    …
    // otros métodos
} // clase Racional
```

150

- La clase Racional ahora permite tres distintas formas de crear instancias:

```
Racional a, b, c;

a = new Racional();      // a = 0/1
b = new Racional(5);     // b = 5/1
c = new Racional(1, 2);  // c = 1/2
```

151

Métodos

- Los métodos contienen el código que comprende y modifica el estado de una instancia.
- Los métodos se invocan como operaciones sobre instancias a través de referencias utilizando el operador punto:

***unObjeto*.*método*(*args* …)**

152

- Añadiendo métodos a la clase Racional:

```
// Método auxiliar para el constructor de
// dos argumentos.
private int máximoComúnDivisor(int x, int y) {
    while (x > 0) {
        if (x < y) {
            int t = x;
            x = y;
            y = t;
        }
        x = x - y;
    }
    return y;
}
```

153

```
public Racional suma(Racional r) {
    int num = numerador * r.denominador
              + denominador * r.numerador;
    int den = denominador * r.denominador;
    return new Racional(num, den);
}

public String toString() {
    return numerador + "/" + denominador;
}
```

154

- Los métodos anteriores se pueden usar así:

```
Racional a = new Racional(1, 2);
Racional b = new Racional(1, 3);
Racional c = a.suma(b);

String s = a.toString();

System.out.println(a);
System.out.println(b);
System.out.println(c);
```

155

La referencia a **`this`**

- La instancia sobre la cual un método es invocado se conoce como el *objeto receptor* del método.
- El objeto receptor está siempre implícito dentro del código del método. Sin embargo, también es accesible de manera explícita utilizando la palabra reservada **`this`**.

156

- Por ejemplo, el método suma de la clase Racional podría ser reescrito así, haciendo uso explícito de **`this`**:

```
public Racional suma(Racional r) {
    int num = this.numerador * r.denominador
              + this.denominador * r.numerador;
    int den = this.denominador * r.denominador;
    return new Racional(num, den);
}
```

157

Sobrecargado de Operaciones

- Dos o más métodos o constructores de una misma clase pueden llamarse igual, siempre y cuando la *firma* de cada método o constructor sea distinta a la del otro que lleva el mismo nombre. A lo anterior se le conoce como *sobrecargado* (*overloading*) de operaciones.

158

- La *firma* de un método o constructor consiste de su nombre y el número y tipo de cada uno de los parámetros formales que recibe.
- El tipo de regreso del método no forma parte de la firma, por lo que no es válido declarar dos métodos que solamente difieran en este aspecto.

159

- Agregando el siguiente método a Racional:

```
public Racional suma(int i) {
    return new Racional(numerador
        + denominador * i, denominador);
}
```

- Ahora se tienen dos versiones distintas de suma, cada una determinable a tiempo de compilación:

```
Racional a, b, c = new Racional(1, 2);
a = c.suma(1);          // a = 3/2
b = c.suma(a);          // b = 2/1
```

160

Herencia

- Toda clase hereda de manera implícita a la clase **Object** predefinida por Java.
- Para que una clase herede de otra clase, se debe utilizar la palabra reservada **`extends`** al momento de declarar la clase:

```
class unaSublase extends unaSuperclase {
    ...
    }
```

161

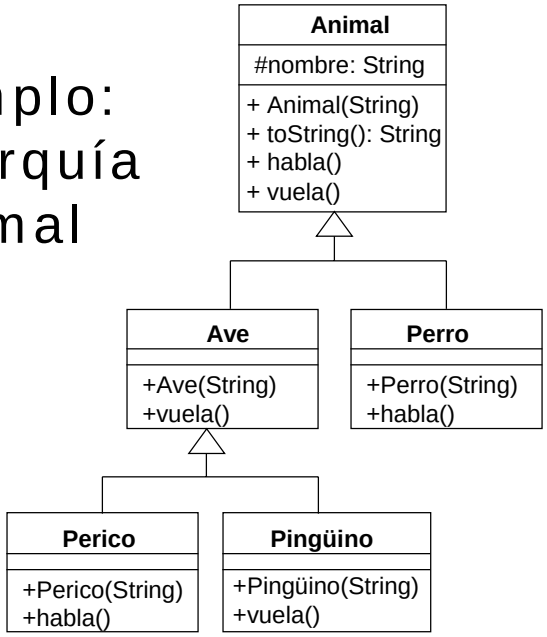
- Java solamente soporta el concepto de *herencia simple*, es decir, solamente se puede heredar directamente de una sola clase.
- La herencia es transitiva, es decir si **A** es una superclase de **B**, y **B** a su vez es una superclase de **C**, entonces **A** es una superclase de **C**.

162

- La subclase hereda el comportamiento y estructura de la superclase. En particular se heredan todas aquellos miembros de las superclases que hayan sido declarados como públicos, protegidos o de paquete.
- Los constructores no se heredan.

163

Ejemplo: Jerarquía Animal



164

- La implementación de las clases quedaría así:

```
class Animal {

    protected String nombre;

    public Animal(String n) {
        nombre = n;
    }

    public String toString() {
        return nombre + " instancia de " + getClass();
    }
```

165

```
    public void habla() {
        System.out.println(toString() + ": no se hablar");
    }

    public void vuela() {
        System.out.println(toString() + ": no se volar");
    }

} // class Animal
```

166

```
class Ave extends Animal {

    public Ave(String n) {
        super(n);
    }

    public void vuela() {
        System.out.println(toString()
            + ": estoy volando");
    }
}
```

167

```
class Perico extends Ave {

    public Perico(String n) {
        super(n);
    }

    public void habla() {
        System.out.println(toString()
            + ": Cuac #$@&*!!!!");
    }
}
```

168

```
class Pinguino extends Ave {

    public Pinguino(String n) {
        super(n);
    }

    public void vuela() {
        System.out.println(toString()
            + ": soy un ave mas no se volar");
    }
}
```

169

```
class Perro extends Animal {

    public Perro(String n) {
        super(n);
    }

    public void habla() {
        System.out.println(toString()
            + ": Gua Gua Gua!");
    }
}
```

170

- Probando las clases:

```
Perico a    = new Perico("Pancho");
Perro b     = new Perro("Firulais");
Pinguino c  = new Pinguino("Polo");
a.habla();
a.vuela();
b.habla();
b.vuela();
c.habla();
c.vuela();
```

171

- La salida del programa anterior sería:

```
Pancho instancia de class Perico: Cuac #$@&*!!!!
Pancho instancia de class Perico: estoy volando
Firulais instancia de class Perro: Gua Gua Gua!
Firulais instancia de class Perro: no se volar
Polo instancia de class Pinguino: no se hablar
Polo instancia de class Pinguino: soy un ave mas no se
volar
```

172

Sobreposición de Métodos

- Si una subclase define un método que tiene la misma firma que un método de alguna de sus superclases, se dice que el método está siendo *redefinido* o *sobrepuesto* (*overriding*).
- Se puede utiliza la palabra reservada **super** dentro del código del método o constructor para invocar a la operación original de la superclase.

173

Conversión por Ensanchamiento

- Sea **A** una clase. Una variable de tipo **A** puede contener en cualquier momento una referencia a una instancia de la clase **A** o a cualquiera de sus subclases sin necesidad de un *cast* explícito.
- A las conversiones por ensanchamiento se les llama también *conversiones seguras* ya que siempre son válidas.

174

- Al ejecutar el siguiente código:

```
Animal x = new Perro("Snoopy");
x.vuela();
x = new Perico("Molly");
x.vuela();
```

se produce la siguiente salida:

```
Snoopy instancia de Perro: no se volar
Molly instancia de Perico: estoy volando
```

175

- Lo anterior es así porque el método que se manda llamar se determina a tiempo de corrida, y está en función del objeto *realmente referido* por la variable y no por el *tipo* de la variable. Este es un ejemplo de polimorfismo.
- Este comportamiento polimórfico se le conoce también como *enlace dinámico*, ya que la referencia al código a ejecutar se resuelve a tiempo de corrida y no de manera estática como ocurre en muchos lenguajes.

176

Operador **instanceof**

- Dado que una variable de la clase **X** puede contener una instancia de **X** o alguna de sus subclases, resulta en ocasiones útil poder determinar exactamente la clase a la que pertenece una determinada instancia.

177

- El operador **instanceof** sirve para el propósito mencionado. La forma de usarse es:

  expresión* instanceof *unaClase

- El operador **instanceof** devuelve el valor booleano **true** si ***expresión*** se refiere a una instancia de ***unaClase*** o alguna de sus subclases, o **false** en caso contrario.

178

- Si ***expresión*** es igual a **`null`**, entonces el operador **`instanceof`** devuelve **`false`**.

```
Animal z = new Pinguino("Marinela");
if (z instanceof Perro)
    System.out.println("Un perro");
if (z instanceof Ave)
    System.out.println("Un ave");
if (z instanceof Pinguino)
    System.out.println("Un pingüino");
```

179

- Sea **`obj`** una variable del tipo **X**, donde **X** es una clase. Para que la expresión

  `obj instanceof C`

  compile, **C** debe ser:

  a) la clase **X**, o

  b) una subclase de la clase **X**, o

  c) una superclase de la clase **X**.

180

- Ejemplos

```
Animal a = new Perro("Brutus");
Perro p = new Perro("Pelusa");

if (a instanceof Animal) {} // válido
if (a instanceof Ave) {}    // válido
if (a instanceof Perro) {}  // válido
if (p instanceof Animal) {} // válido
if (p instanceof Ave) {}    // inválido
if (p instanceof Perro) {}  // válido
```

181

Conversión por Estrechamiento

- Sea **B** una subclase de **A**. Una variable de tipo **A** puede ser convertida a un tipo **B** a través de un *cast* explícito.
- Dicha conversión es válida sólo si la instancia referida por la variable de tipo **A** es, a tiempo de corrida, una instancia de tipo **B** o una de sus subclases. De no ser así, se arroja una **ClassCastException** a tiempo de corrida.

182

- Ejemplos:

```
Animal a = new Perro("Dogbert");
Animal b = new Perico("Panfilo");

Perro c = (Perro) a;       // bien
Ave d = (Ave) b;           // bien
Perico e = (Perico) b;     // bien
Ave f = (Ave) a;           // error
Pinguino g = (Pinguino) b; // error
Perro h = (Perro) b;       // error
```

183

- Se puede utilizar **`instanceof`** antes de hacer el cast para evitar hacer una conversión inválida:

```
Perro h;
if (b instanceof Perro)
    h = (Perro) b;
```

- A las conversiones por estrechamiento se les llama también *conversiones inseguras* ya que pueden llegar a ser inválidas.

184

Variables Estáticas

- Si un campo de una clase se declara **`static`**, significa que solamente existe una variable para toda la clase, sin importar cuantas instancias (incluso cero) de la clase sean eventualmente creadas.
- A las variables estáticas también se les conoce como *variables de clase*.
- La forma genérica de acceder a una variable de clase es:

  `nombreClase.variableEstática`

185

Datos Constantes

- Para declarar un miembro de una clase como constante, se utilizan las palabras reservadas **`static`** y **`final`**. El indicar que una variable es **`final`** implica que su valor no puede cambiar, haciéndola realmente una constante.
- Por ejemplo:

  ```
  public static final int MAX = 50;
  ```

- Las constantes deben ser inicializadas durante su declaración.

186

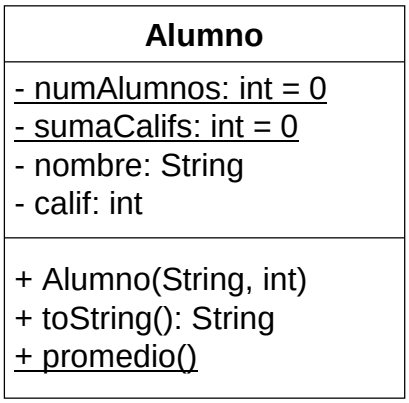
Métodos Estáticos

- Un método estático es un *método de clase*.
- Un método de clase se invoca sin una referencia a un objeto en particular.
- Es un error intentar acceder a **`this`** o **`super`** dentro del cuerpo de un método de clase.
- La forma genérica de invocar un método de clase es:

```
nombreClase.métodoEstático(args …)
```

187

Ejemplo de Uso de Miembros Estáticos



188

```
class Alumno {

    // Variables de clase
    private static int numAlumnos = 0;
    private static int sumaCalifs = 0;

    // Variables de instancia
    private String nombre;
    private int calif;
```

189

```
    public Alumno(String n, int c) {
        nombre = n;
        calif = c;
        numAlumnos ++;
        sumaCalifs += calif;
    }

    public String toString() {
        return nombre + ", calificación: "
            + calif;
    }

    public static double promedio() {
        return (double) sumaCalifs / numAlumnos;
    }
} // class Alumno
```

190

- Utilizando la clase anterior:

```
Alumno[] a = { new Alumno("Pepe", 69),
               new Alumno("Cuca", 76),
               new Alumno("Chuy", 92) };

for (int i = 0; i < a.length; i ++)
    System.out.println(a[i].toString());

System.out.println("Promedio del grupo: "
    + Alumno.promedio());
```

produce la siguiente salida:

```
Pepe, calificación: 69
Cuca, calificación: 76
Chuy, calificación: 92
Promedio del grupo: 79.0
```

191

Clases Abstractas

- Una clase abstracta es aquella que no puede ser instanciable.
- Normalmente una clase abstracta no implementa todos los métodos que declara, sino que delega esta responsabilidad a sus subclases.

192

- Para indicar que una clase es abstracta, en su declaración se incluye la palabra reservada **`abstract`**:

```
abstract class unaClase
```

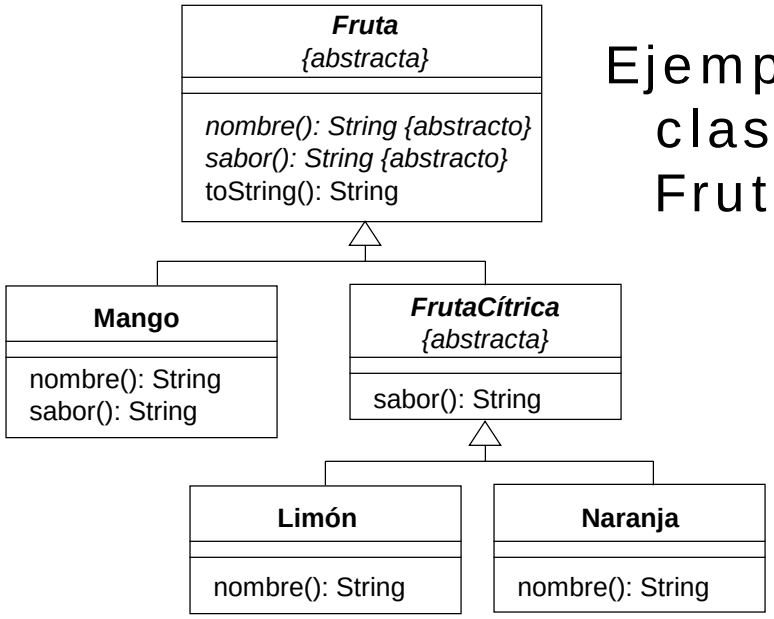
- Un método abstracto es un método de una clase abstracta que no contiene código. Su declaración es de la forma:

```
public abstract tipo método(args ...);
```

193

- El método abstracto deberá ser implementado por la clase concreta que herede de la clase abstracta en cuestión.
- Un método abstracto no puede ser de tipo **`private`**.
- Sea **X** una clase abstracta. A pesar de que no se pueden crear instancias de **X**, sí pueden haber variables de tipo **X**, las cuales podrán contener referencia a instancias de subclases concretas de **X**.

194



Ejemplo: clase Fruta

195

```
abstract class Fruta {
    public abstract String nombre();
    public abstract String sabor();

    public String toString() {
        return "Soy " + nombre()
            + " y mi sabor es " + sabor();
    }
}

abstract class FrutaCitrica extends Fruta {
    public String sabor() {
        return "ácido";
    }
}
```

196

```
class Limon extends FrutaCitrica {
    public String nombre() {
        return "limón sin semilla";
    }
}

class Naranja extends FrutaCitrica {
    public String nombre() {
        return "naranja para jugo";
    }
}
```

197

```
class Mango extends Fruta {
    public String nombre() {
        return "mango manila";
    }

    public String sabor() {
        return "dulce";
    }
}
```

198

- Demostrando el uso de las clases anteriores:

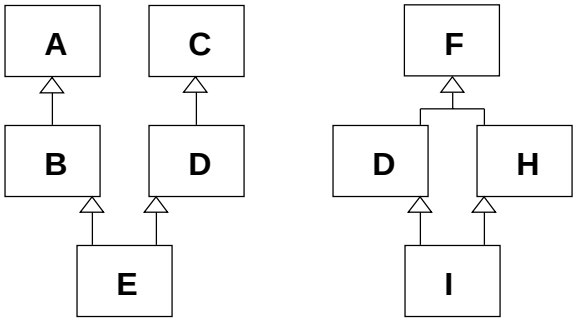
```
Fruta[] f = { new Limon(),
              new Mango(),
              new Naranja() };

for (int i = 0; i < f.length; i ++)
    System.out.println(f[i].toString());
```

Se produce la siguiente salida:

```
Soy limón sin semilla y mi sabor es ácido
Soy mango manila y mi sabor es dulce
Soy naranja para jugo y mi sabor es ácido
```

199

Herencia múltiple



- Cuando una clase hereda de dos o más clase se le llama *herencia múltiple*.

200

- ¿Qué pasa con la clase **E** si la clase **A** y **D** tienen un método con la misma firma? ¿Hereda uno de los dos métodos? ¿Surge una ambigüedad?
- ¿Qué pasa con la clase **I** que hereda de la clase **F** por dos caminos? ¿Se duplican sus variables de instancia?
- Estos son los problemas que se quisieron evitar en Java al solo permitir herencia de simple.

201

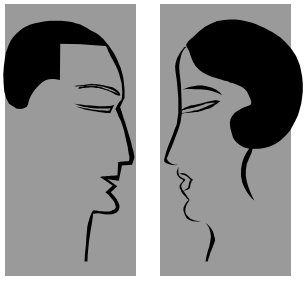
- No obstante a estos problemas, la herencia múltiple es un concepto útil, por lo que Java provee de un mecanismo llamado *interfaces* en donde se obtienen algunos de los beneficios de la herencia múltiple pero sin sus complicaciones.

202

CAPÍTULO 6

Interfaces

Interfaces



- Una interface es un tipo cuyos miembros son constantes y métodos abstractos. En muchos aspectos es similar a una clase abstracta.
- Todos los métodos de una interface son *públicos* y *abstractos*.
- Todos los campos de una interface son *públicos*, *estáticos* y *finales*, es decir son constantes.

204

- El uso de interfaces en Java hace que sea innecesario que clases relacionadas tengan que compartir una misma superclase abstracta, o que se tengan que añadir métodos a la clase **Object**.
- Una clase puede heredar de otra única clase, sin embargo puede implementar múltiples interfaces.

205

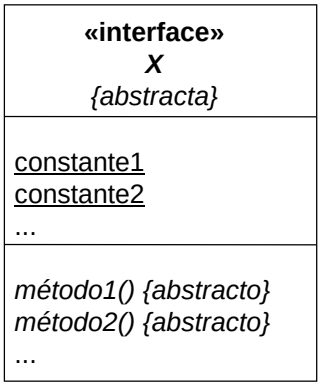
- Colectivamente a todas las *superclases* y *superinterfaces* de una clase reciben el nombre de *supertipos*.
- Una subclase hereda la implementación de su superclase.
- Sin embargo una clase que implementa una interface se dice que hereda precisamente la “interface” de su superinterface. La implementación corre por cuenta de dicha clase, ya que todos los métodos de la superinterface son abstractos.

206

- Se dice que Java soporta *herencia simple de implementación* (mediante el mecanismo de herencia de clases) pero que soporta *herencia múltiple de interfaces*.
- Este tipo de herencia múltiple de interfaces permite que los objetos soporten múltiples comportamientos comunes sin tener que compartir la implementación.

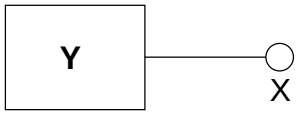
207

- En UML una interface se especifica como una clase abstracta pero añadiendo el estereotipo «interface».



208

- Para indicar que una clase implementa una interface, se utiliza la siguiente notación (la relación siempre es de 1 a 1):



La clase **Y** implementa la interface **X**.

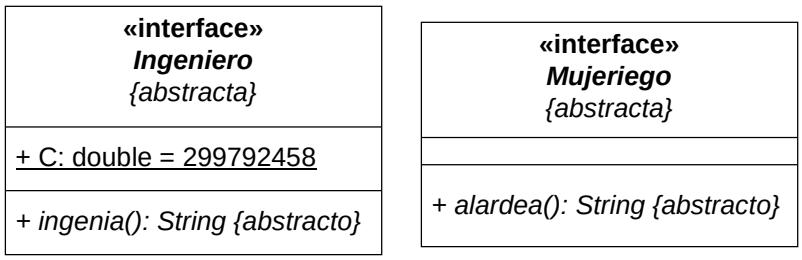
209

Definiendo una Interface

- La definición de una interface es parecida a la de una clase. En lugar de utilizar la palabra reservada **`class`** se utiliza **`interface`**.
- Los métodos de una interface, por ser abstractos, no deben contener cuerpo.
- Los campos, por ser finales y estáticos, deben estar inicializados.

210

Ejemplo: Ingeniero y Mujeriego



211

```
interface Ingeniero {
    double C = 299792458;
    String ingenia();
}

interface Mujeriego {
    String alardea();
}
```

212

Implementando una Interface

- Una clase puede implementar una o más interfaces. La palabra reservada **`implements`** se utiliza para este efecto. Por ejemplo:

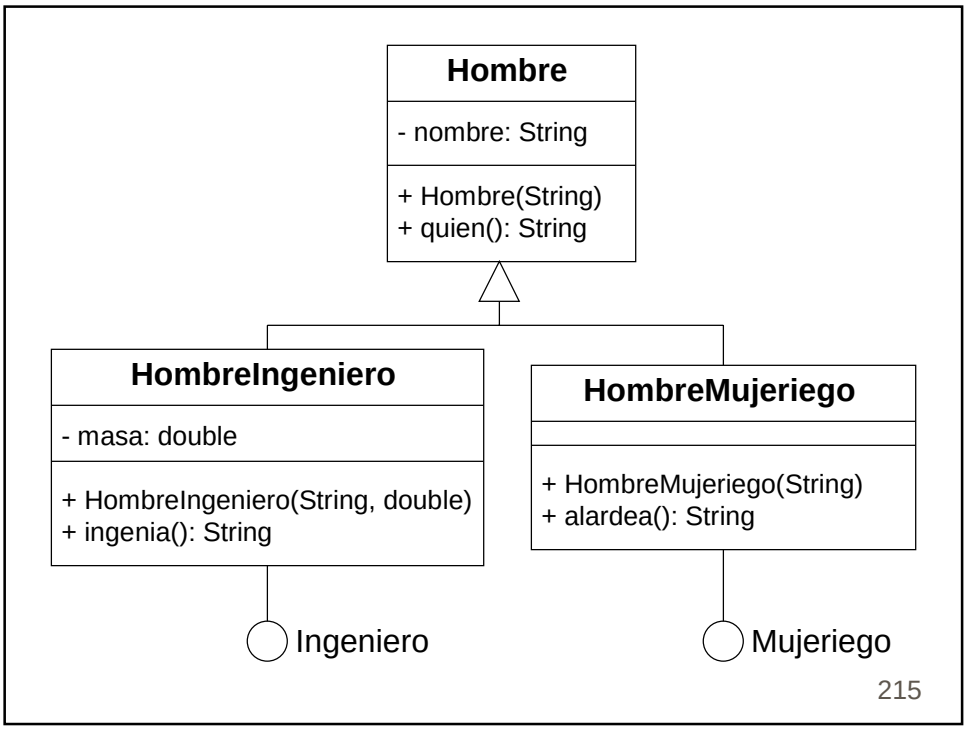
```
class A implements B, C, D {
    ...
}
```

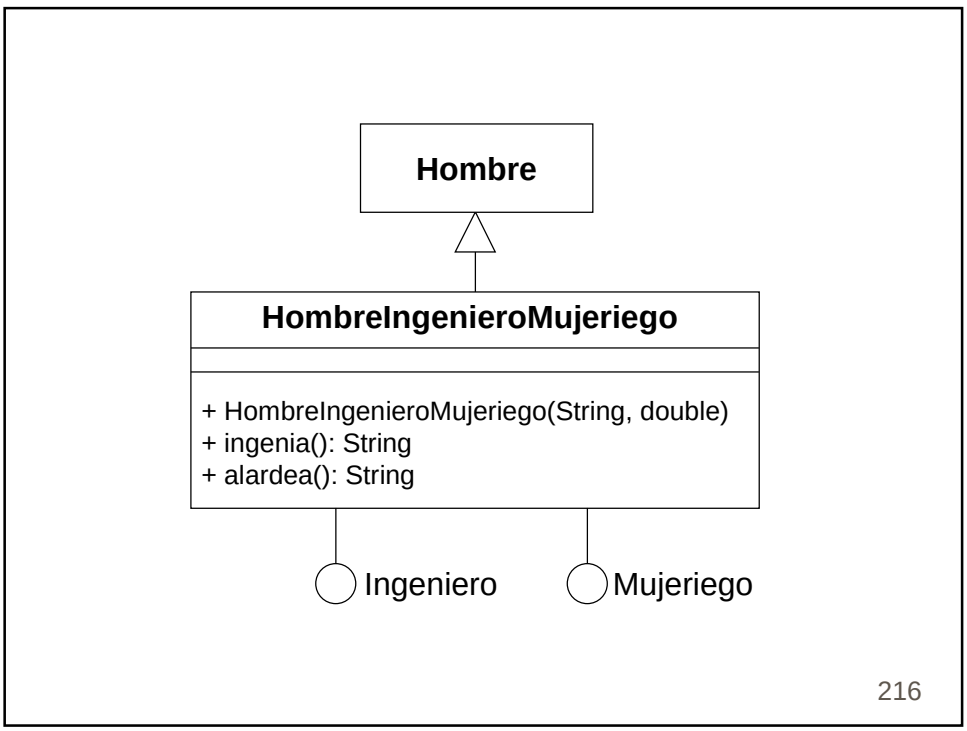
define la clase **A**, la cual implementa las interfaces **B**, **C** y **D**.

213

- Si la clase **A** implementa la interface **B**, es responsabilidad de **A** proporcionar la implementación de los métodos abstractos de **B**, a menos que **A** sea una clase abstracta, en cuyo caso la implementación corre a cargo de las subclases de **A**.
- Colectivamente a las variables cuyo tipo es una clase o una interface se les llama de *tipo referencia*.

214





```
class Hombre {
    private String nombre;

    public Hombre(String n) {
        nombre = n;
    }

    public String quien() {
        return nombre;
    }
}
```

217

```
class HombreIngeniero extends Hombre
    implements Ingeniero {

    private double masa;

    public HombreIngeniero(String n, double m) {
        super(n);
        masa = m;
    }

    public String ingenia() {
        return "Mi energía es = "
            + (masa * C * C);
    }
}
```

218

```
class HombreMujeriego extends Hombre
    implements Mujeriego {

    public HombreMujeriego(String n) {
        super(n);
    }

    public String alardea() {
        return "Tengo muchas viejas.";
    }
}
```

219

```
class HombreIngenieroMujeriego extends Hombre
    implements Ingeniero, Mujeriego {

    public HombreIngenieroMujeriego(String n) {
        super(n);
    }

    public String ingenia() {
        return "Delta = (Num. Viejas "
            + "Esta Semana) - (Num. Viejas "
            + "Semana Pasada)";
    }

    public String alardea() {
        return "Mi Delta siempre es positiva.";
    }
}
```

220

Interfaces y el Operador **`instanceof`**

- Es posible determinar a tiempo de corrida si un objeto (sin importar su tipo) es una instancia de una clase que implementa una interface utilizando el operador **`instanceof`**.

221

- El siguiente código

```
Hombre[] h =
    { new Hombre("Sultano"),
      new HombreIngeniero("Petronilo", 65.2),
      new HombreMujeriego("Espiridiono"),
      new HombreIngenieroMujeriego("Epitacio") };

for (int i = 0; i < h.length; i ++) {
    System.out.println("Sr. " + h[i].quien());
    if (h[i] instanceof Ingeniero)
        System.out.println("Ing. " + h[i].quien()
            + ": " + ((Ingeniero) h[i]).ingenia());
    if (h[i] instanceof Mujeriego)
        System.out.println("Don " + h[i].quien()
            + " Tenorio: "
            + ((Mujeriego) h[i]).alardea());
    System.out.println();
}
```

222

genera esta salida:

```
Sr. Sultano

Sr. Petronilo
Ing. Petronilo: Mi energía es = 5.859883765364052E18

Sr. Espiridiono
Don Espiridiono Tenorio: Tengo muchas viejas.

Sr. Epitacio
Ing. Epitacio: Delta = (Num. Viejas Esta Semana) -
(Num. Viejas Semana Pasada)
Don Epitacio Tenorio: Mi Delta siempre es positiva.
```

223

Variables de Tipo Interface

- Una variable de tipo interface puede contener cualquier instancia de una clase que implemente dicha interface.
- Las expresiones de tipo interface pueden ser convertidas a otros tipos a través de conversiones de estrechamiento y ensanchamiento.

224

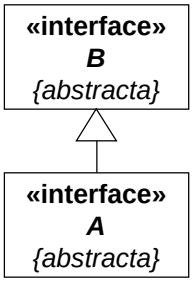
- Por ejemplo:

```
HombreIngenieroMujeriego him =
  new HombreIngenieroMujeriego("Panfilo");
Hombre    h = him;  // conversión por ensanchamiento
Ingeniero i = him;  // conversión por ensanchamiento
Mujeriego m = him;  // conversión por ensanchamiento

System.out.println(h.quien());
System.out.println(i.ingenia());
System.out.println(m.alardea());

// Requieren conversión por estrechamiento
System.out.println(((Hombre) i).quien());
System.out.println(((Ingeniero) m).ingenia());
System.out.println(((Mujeriego) h).alardea());
```

225

Extendiendo Interfaces

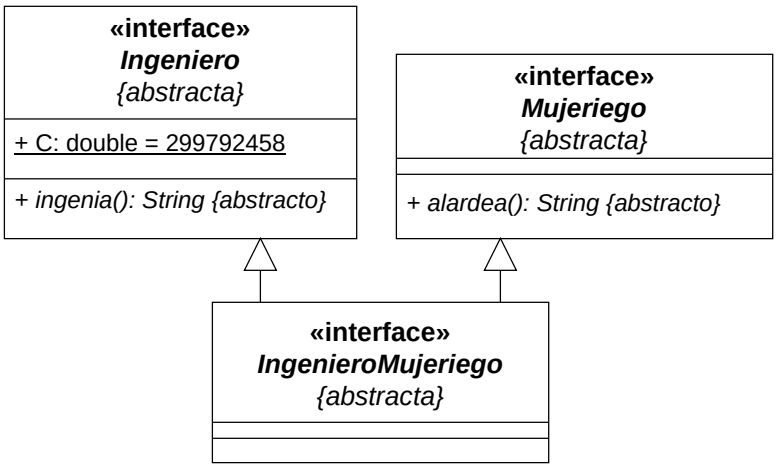
- Una interface puede extender una o más interfaces, heredando de esta forma las constantes y los métodos abstractos de dichas interfaces. Por ejemplo:



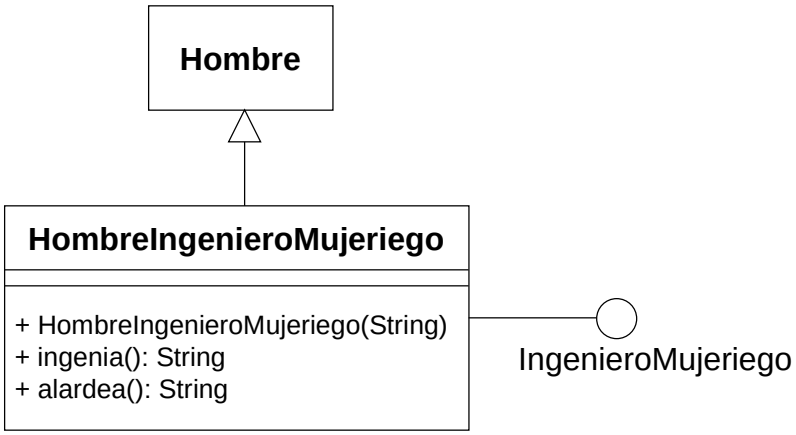
B es una *superinterface* de A.

226

Ejemplo: Interface IngenieroMujeriego



227



228

- La interface **IngenieroMujeriego** no añade ningún método o constante nueva, pero hereda los métodos y constantes de sus superinterfaces.

```
interface IngenieroMujeriego
    extends Ingeniero, Mujeriego {
}
```

- Cualquier clase que implemente la interface **IngenieroMujeriego** es responsable también de implementar los métodos de las superinterfaces de ésta.

229

```
class HombreIngenieroMujeriego extends Hombre
    implements IngenieroMujeriego {

    public HombreIngenieroMujeriego(String n) {
        super(n);
    }

    public String ingenia() {
        /* ... */
    }

    public String alardea() {
        /* ... */
    }
}
```

230

Colisión entre Nombres de Constantes

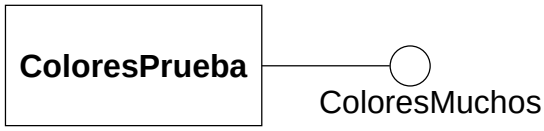
- Es posible que una interface herede uno o más campos con el mismo nombre. Esta situación por sí sola no produce un error de compilación.

231

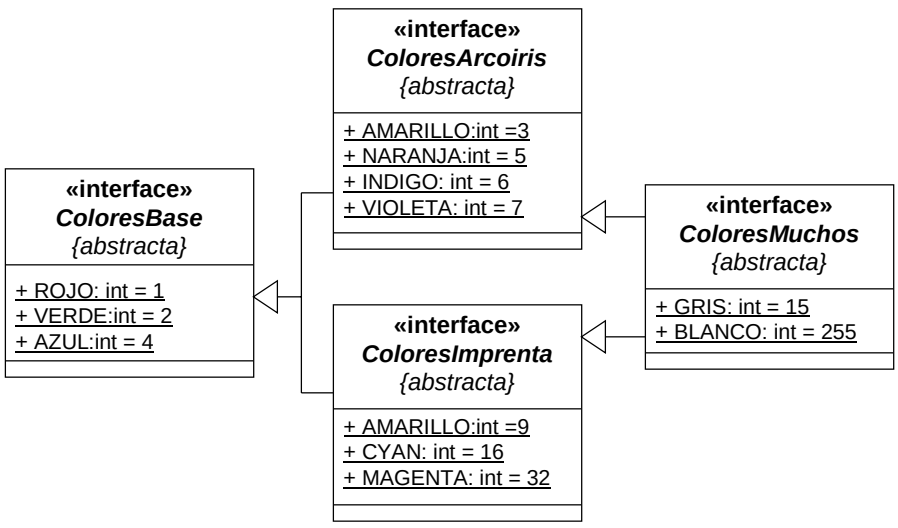
- Sin embargo, el referirse a dicho campo por su nombre simple se considera una referencia ambigua y sí es motivo para generar un error de compilación.
- Pueden existir varias rutas por las que se puedan heredar un mismo campo. En dicha situación el campo se considera heredado una sola vez, y puede ser referido por su nombre simple sin ambigüedad.

232

Ejemplo: Interfaces de Colores



233



234

```
interface ColoresBase {
    int ROJO      =   1;
    int VERDE     =   2;
    int AZUL      =   4;
}

interface ColoresArcoiris
    extends ColoresBase {
    int AMARILLO  =   3;
    int NARANJA   =   5;
    int INDIGO    =   6;
    int VIOLETA   =   7;
}
```

235

```
interface ColoresImprenta
    extends ColoresBase {
    int AMARILLO  =   3;
    int CYAN      =  16;
    int MAGENTA   =  32;
}

interface ColoresMuchos
    extends ColoresArcoiris, ColoresImprenta {
    int GRIS      =  15;
    int BLANCO    = 255;
}
```

236

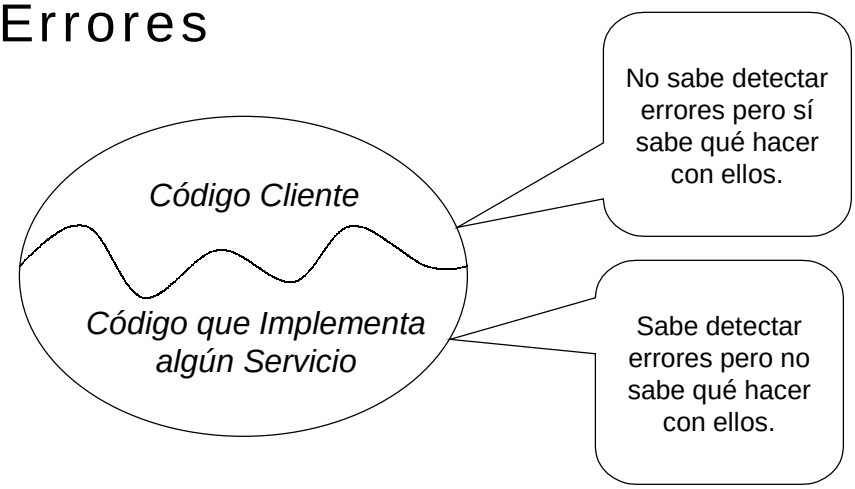
```
class ColoresPrueba implements ColoresMuchos {
    public static void main(String[] args) {
        int a = CYAN + GRIS;          // Bien
        int b = INDIGO + BLANCO;      // Bien
        int c = AMARILLO + BLANCO;    // Ambiguo
        int d = ColoresArcoiris.AMARILLO
            + ColoresImprenta.AMARILLO; // Bien
        int e = ROJO + AZUL;          // Bien
    }
}
```

237

CAPÍTULO 7

Manejo de Excepciones

Detección de Errores



239

¿Qué Hacer Cuando el Código que Implementa Algún Servicio Detecta un Error?

- Abortar el programa.
- Devolver un valor especial.
- Arrojar una excepción.

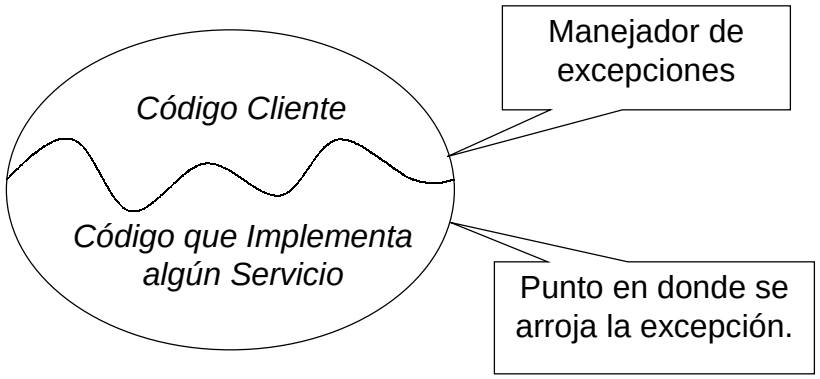
240

Excepciones

- Una *excepción* es un evento que ocurre durante la ejecución de un programa que altera el flujo normal de instrucciones.
- Se dice que una *excepción* es arrojada en el punto en el que ocurre y es *atrapada* en el punto en el que se transfiere el control.

241

Manejo de Excepciones



242

Sentencia **`try/catch/finally`**

- Cuando se arroja una excepción, se transfiere el control a la cláusula **`catch`** del manejador de excepciones dinámicamente más cercano que maneje la excepción correspondiente.

243

- Una cláusula **`catch`** atrapará un objeto excepción que sea una instancia (según las reglas de **`instanceof`**) del tipo del parámetro declarado.
- La transferencia de control que ocurre cuando se arroja una excepción produce la *terminación abrupta* de expresiones y sentencias hasta que se encuentre una cláusula **`catch`** capaz de manejarla.

244

- La ejecución continúa con el código asociado a dicha cláusula **`catch`**. Una vez concluido éste, el flujo prosigue con las sentencias que se encuentran después de toda la sentencia **`try/catch`**.
- El código que causó la excepción nunca es resumido.

245

- Si se desea garantizar que un bloque de código siempre sea ejecutado después de otro, incluso cuando ese otro bloque haya terminado de manera abrupta, se puede utilizar una sentencia **`try`** con una cláusula **`finally`**.
- La cláusula **`finally`** siempre se ejecuta, sin importar que el bloque del **`try`** haya o no terminado de manera abrupta, o si se haya o no atrapado una excepción en alguna de la cláusulas **`catch`**.

246

```
class EjemExcep {

    private static int acumulador = 0;
    private static int contador = 0;

    public static void main(String[] args) {
        while (contador < 4) {
            try {
                metodo1();
                acumulador ++;
            } catch (NullPointerException e) {
                System.out.println(e);
                acumulador ++;
            }
```

247

```
            catch (Exception e) {
                System.out.println(e);
                acumulador ++;
            }
            contador ++;
        } // while
        System.out.println(acumulador);
    } // main
```

248

```
private static void metodo1() {
    if (contador == 1) {
        metodo2();
        acumulador ++;
    } else {
        try {
            metodo2();
            acumulador ++;
        } catch (NegativeArraySizeException e) {
            System.out.println(e);
            acumulador ++;
        }
```

249

```
        catch (IndexOutOfBoundsException e) {
            System.out.println(e);
            acumulador ++;
        } finally {
            acumulador ++;
        }
        acumulador ++;
    } // else
    acumulador ++;
} // metodo1
```

250

```
    private static void metodo2() {
        int[] a;
        switch (contador) {
        case 0:
            a = null;
            a[0] = 0;
            break;

        case 1:
            a = new int[-1];
            break;
```

251

```
        case 2:
            a = new int[1];
            a[1] = 1;
            break;

        case 3:
            int x = 0, y = 1 / x;
            break;
        }
        acumulador ++;
    } // metodo2

} // class EjemExcep
```

252

- La salida del programa anterior sería:

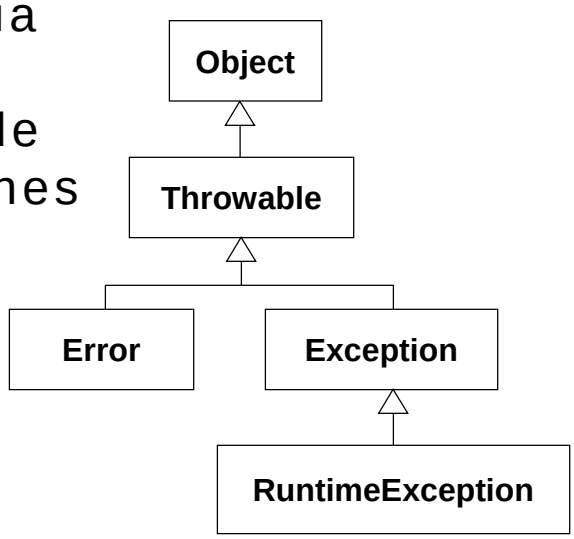
```
java.lang.NullPointerException:
java.lang.NegativeArraySizeException:
java.lang.ArrayIndexOutOfBoundsException: 1
java.lang.ArithmeticException: / by zero
10
```

253

- Si no existe ninguna cláusula catch que atrape a la excepción arrojada, entonces el *thread* en ejecución en abortado y se imprime en la salida de error estándar el rastro de ejecución hasta llegar al punto en donde ocurrió la excepción.
- Dentro de una cláusula **`catch`** solamente se puede declarar un parámetro de tipo **Throwable** o alguna de sus subclases.

254

Jerarquía de las Clases de Excepciones



255

- La clase **Error** y sus subclases son excepciones de las que normalmente no se espera que un programa se recupere. Por ejemplo: **OutOfMemoryError**, **StackOverflowError** y **ClassFormatError**.
- La clase **Exception** es la superclase de todas las excepciones de las que normalmente un programa desea recuperarse.

256

- La clase **RuntimeException** y sus subclases sirven para distinguir varios tipos de errores que pueden ocurrir a tiempo de corrida. Por ejemplo: **ArithmeticException**, **NullPointerException** e **IndexOutOfBoundsException**.

257

Verificación de Excepciones

Una excepción puede ser

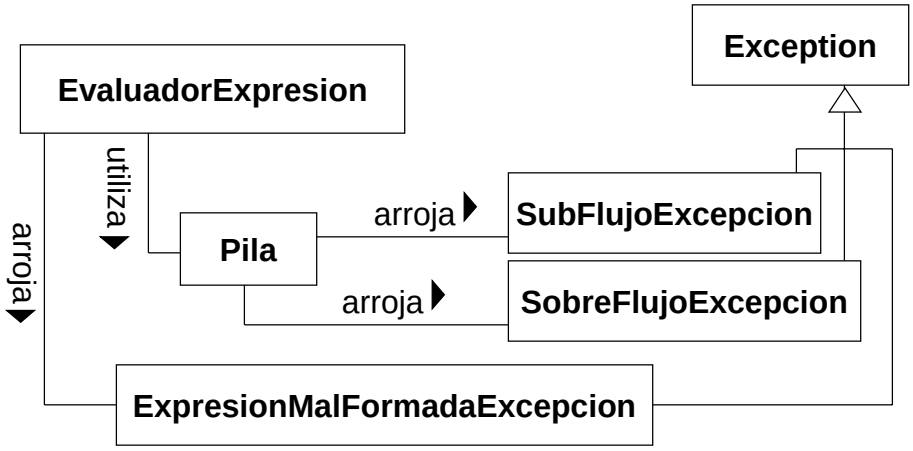
- ***Verificada***, en cuyo caso el compilador espera que la excepción sea:
  - ✔ Atrapada en a través de una cláusula **`catch`**.
  - ✔ Vuelta a arrojar mediante su declaración en la cláusula **`throws`** de el método o constructor en que aparece.
- ***No verificada***, en cuyo caso el compilador no no realizar ninguna validación.

258

- La clase **Error** y sus respectivas subclases son excepciones no verificadas ya que potencialmente pueden ocurrir en muchos puntos del programa y su recuperación es difícil o imposible.
- La clase **RuntimeException** y sus subclases tampoco son verificadas por el compilador de Java ya que representan errores que normalmente son evitados por un programa bien escrito.
- Todas las demás excepciones son excepciones verificadas.

259

Ejemplo: Clase EvaluadorExpresion



260

```
class Pila {

    private int[] info;
    private int nivel;

    public Pila(int n) {
        nivel = 0;
        info = new int[n];
    }

    public boolean vacia() {
        return nivel == 0;
    }
```

261

```
   public void push(int dato)
            throws SobreFlujoExcepcion {
       if (nivel == info.length) {
           throw new SobreFlujoExcepcion(
               "Pila desbordada");
       }
       info[nivel] = dato;
       nivel ++;
   }
```

262

```
    public int pop()
            throws SubFlujoExcepcion {
        if (vacia()) {
           throw new SubFlujoExcepcion(
               "No se puede sacar un elemento"
               + "de una pila vacía");
        }
        nivel --;
        return info[nivel];
    }
} // class Pila
```

263

```
class SobreFlujoExcepcion extends Exception {
    public SobreFlujoExcepcion(String mensaje) {
        super(mensaje);
    }
}

class SubFlujoExcepcion extends Exception {
    public SubFlujoExcepcion(String mensaje) {
        super(mensaje);
    }
}
```

264

```
class ExpresionMalFormadaExcepcion
    extends Exception {

    public ExpresionMalFormadaExcepcion(
            String mensaje) {
        super(mensaje);
    }
}
```

265

```
class EvaluadorExpresion {

    private String entrada;

    public EvaluadorExpresion(String s) {
        entrada = s;
    }

    public int evalua()
            throws ExpresionMalFormadaExcepcion {
        Pila p = new Pila(5);
        int i;
```

266

```
try {
    for (i = 0; i < entrada.length(); i ++) {
        char c = entrada.charAt(i);
        if (Character.isDigit(c)) {
            p.push(Character.digit(c, 10));
        } else {
            int op2 = p.pop();
            int op1 = p.pop();
            switch (c) {
            case '+':
                p.push(op1 + op2);
                break;
            case '-':
                p.push(op1 - op2);
                break;
```

267

```
            case '*':
                p.push(op1 * op2);
                break;
            case '/':
                p.push(op1 / op2);
                break;
            default:
                throw new
                    ExpresionMalFormadaExcepcion(
                        "Carácter \'" + c
                        + "\' inválido en "
                        + entrada);
                break;
            } // switch
        } // else
    } // for
```

268

```
    int resultado = p.pop();

    if (! p.vacia()) {
        throw new
            ExpresionMalFormadaExcepcion(
            "Insufucientes operadores en "
            + entrada);
    }
    return resultado;

  } catch (SubFlujoExcepcion e) {
      throw new ExpresionMalFormadaExcepcion(
          "Insuficientes operandos en "
          + entrada);
  }
```

269

```
    catch (SobreFlujoExcepcion e) {
        throw new ExpresionMalFormadaExcepcion(
            "Pila muy pequeña para evaluar "
            + "expresión " + entrada);
    }
} // evalua
```

270

```
    public static void main(String[] args)
          throws ExpresionMalFormadaExcepcion {
        if (args.length != 1) {
            System.err.println(
                "Número incorrecto de "
                + "argumentos");
            System.exit(1);
        }
        System.out.println(
            new EvaluadorExpresion(
                args[0]).evalua());
    } // main
} // class EvaluadorExpresion
```

271

Ventajas del Manejo de Excepciones

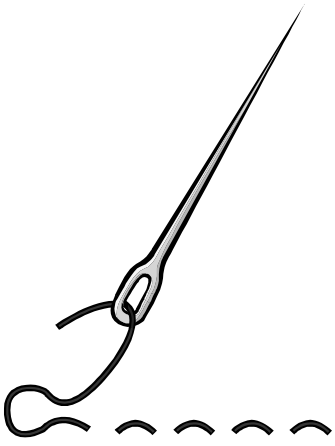
- Separación del código de manejo de errores y el código “regular”.
- Propagación de errores a través de la pila del sistema.
- Capacidad para agrupar y diferenciar los tipos de errores.

272

CAPÍTULO 8

Threads

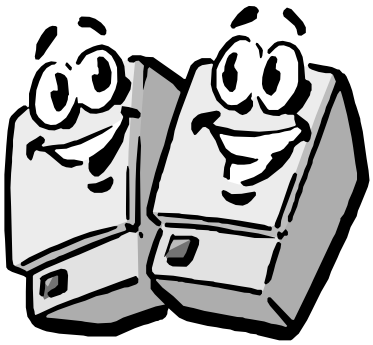
Threads



- Porción de un programa que lleva a cabo una tarea.
- A un *thread* se le conoce también como un hilo, hebra, subproceso o proceso ligero.
- Un programa concurrente es aquel en que existen varios *threads* ejecutándose al mismo tiempo.

274

Implementación de Concurrencia



- **Multiprogramación:** Un procesador ejecuta varios *threads* al mismo tiempo, ya sea de manera cooperativa o preemptiva.
- **Multiprocesamiento:** Potencialmente, cada *thread* se ejecuta en un procesador independiente.

275

Opciones para usar *threads* en Java

- Extender la clase Thread y sobreponer el método **`run()`**.
- Implementar la interface Runnable junto con su método **`run()`**.

276

Extendiendo Thread

```
class PinPonPapas extends Thread {
    private String palabra;  // Palabra a imprimir.
    private int retardo;   // Retardo entre impresiones.

    public PinPonPapas(String s, int i) {
        palabra = s;
        retardo = i;
    }
```

277

```
       public void run() {
           try {
               for (int i = 0; i < 10; i ++) {
                   System.out.print(palabra + " ");
                   Thread.sleep(retardo);
               }
           } catch(InterruptedException e) {}
       }
   } // class PinPonPapas
```

278

Usando los *threads*

```
PinPonPapas t1, t2, t3;
t1 = new PinPonPapas("pin", 33);
t2 = new PinPonPapas("pon", 66);
t3 = new PinPonPapas("papas", 100);

t1.start();
t2.start();
t3.start();
```

279

Implementando Runnable

```
class PinPonPapas2 implements Runnable {
    private String palabra;  // Palabra a imprimir.
    private int retardo;   // Retardo entre impresiones.

    public PinPonPapas2(String s, int i) {
        palabra = s;
        retardo = i;
    }
```

280

```
    public void run() {
        try {
            for (int i = 0; i < 10; i ++) {
                System.out.print(palabra + " ");
                Thread.sleep(retardo);
            }
        } catch(InterruptedException e) {}
    }
} // class PinPonPapas2
```

281

Usando los *threads*

```
PinPonPapas2 pin = new PinPonPapas2("pin", 33);
PinPonPapas2 pon = new PinPonPapas2("pon", 66);
PinPonPapas2 papas = new PinPonPapas2("papas", 100);

Thread t1 = new Thread(pin);
Thread t2 = new Thread(pon);
Thread t3 = new Thread(papas);

t1.start();
t2.start();
t3.start();
```

282

Problemas con la Concurrencia

- **Exclusión Mutua** Dos o más *threads* pueden acceder a un mismo recurso al mismo tiempo. Se busca que no se produzcan inconsistencias.
- **Comunicación** Un *thread* debe esperar a que otro *thread* termine cierta tarea antes de continuar.

283

Synchronized

- Sirve para garantizar exclusión mutua.
- Un método se puede declarar **`synchronized`** para obtener un candado sobre el objeto receptor. En ese caso el *thread* actual posee el candado de dicho objeto. Ningún otro *thread* puede ejecutar un método sincronizado sobre dicho objeto.
- Dicho candado se libera hasta que el método termina.

284

Wait y Notify

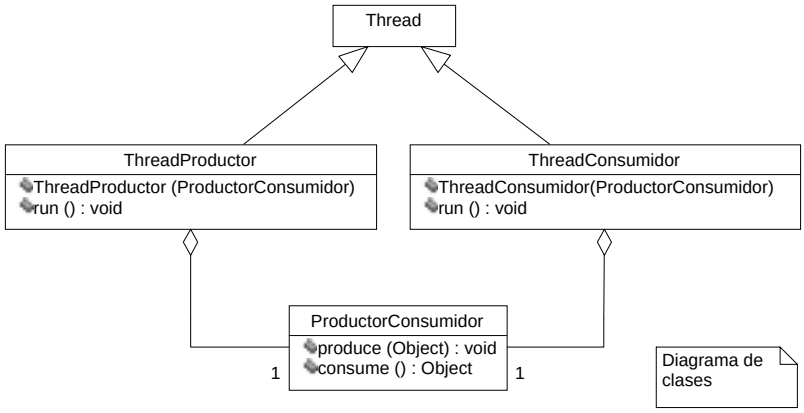
- Permite a dos threads comunicarse.
- Los métodos **`wait()`** y **`notify()`** son métodos de la clase Object.
- Ambos métodos deben ser ejecutados dentro de un método sincronizado.
- Un *thread* lleva a cabo un **`wait()`** sobre un cierto objeto para suspenderse hasta que una determinada condición se cumpla.

285

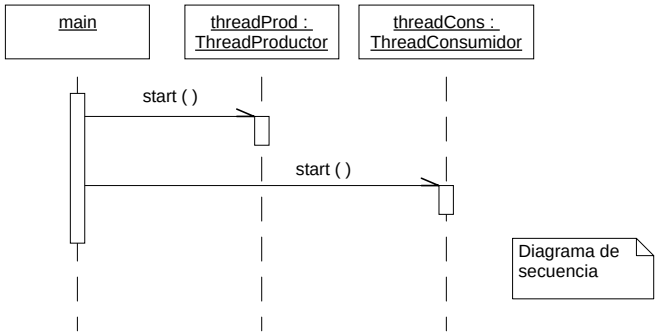
- Otro *thread* hace un **`notify()`** sobre el mismo objeto, para indicarle al primer *thread* que la condición posiblemente ya fue cumplida.
- Cuando ocurre un **`wait()`**, de manera atómica el *thread* actual es suspendido y el candado del objeto correspondiente es liberado.
- El **`wait()`** debe estar en un ciclo para tener la certeza de que fue notificado debido a que la condición en cuestión fue cumplida.

286

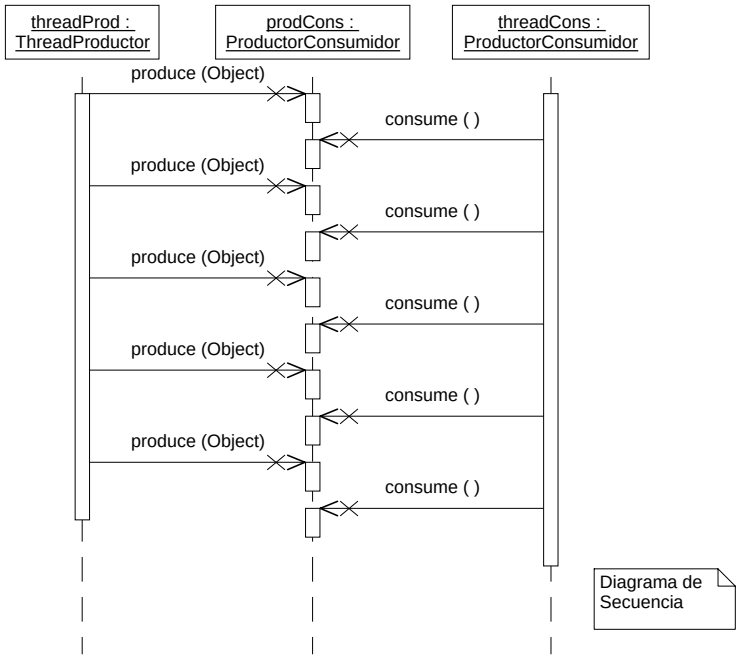
Ejemplo: Productor-Consumidor



287



288



289

```
class ProductorConsumidor {
    private Object valor = null;

    public synchronized void produce(Object obj) {
        try {
            while (valor != null) {
                wait();
            }
        } catch(InterruptedException e) {
            return;
        }

        valor = obj;
        notifyAll();
    }
```

290

```
       public synchronized Object consume() {
           try {
               while (valor == null) {
                   wait();
               }
           } catch(InterruptedException e) {
               return null;
           }

           Object obj = valor;
           valor = null;
           notifyAll();
           return obj;
       }
   } // class ProductorConsumidor
```

291

```
class ThreadProductor extends Thread {
    ProductorConsumidor prodCons;

    public ThreadProductor(ProductorConsumidor pc) {
        prodCons = pc;
    }

    public void run() {
        System.out.println("Comienza thread productor");
        for (int i = 0; i < 5; i ++) {
            int x = (int) (Math.random() * 1000);
            System.out.println("Numero producido: " + x);
            prodCons.produce(new Integer(x));
        }
        System.out.println("Termina thread productor");
    }
}
```

292

```
class ThreadConsumidor extends Thread {
    ProductorConsumidor prodCons;

    public ThreadConsumidor(ProductorConsumidor pc) {
        prodCons = pc;
    }

    public void run() {
        System.out.println("Comienza thread consumidor");
        for (int i = 0; i < 5; i ++) {
            Object obj = prodCons.consume();
            System.out.println("Numero consumido: "
                + obj);
        }
        System.out.println("Termina thread consumidor");
    }
}
```

293

Probando el Productor-Consumidor

```
System.out.println("Comienza thread principal");

ProductorConsumidor prodCons =
    new ProductorConsumidor();

ThreadProductor threadProd =
    new ThreadProductor(prodCons);
ThreadConsumidor threadCons =
    new ThreadConsumidor(prodCons);

threadProd.start();
threadCons.start();

System.out.println("Termina thread principal");
```

294

Operaciones de Interés

- **sleep:** Método estático de Thread que pone al *thread* actual a dormir por un determinado tiempo.
- **yield:** Método estático de Thread que cede el control del *thread* actual para que pueda ejecutarse algún otro *thread*.
- **currentThread:** Método estático de Thread que devuelve una referencia al objeto que representa el *thread* en ejecución.

295

- **suspend:** Suspende temporalmente la ejecución de un *thread*.
- **resume:** Continua la ejecución de un *thread* previamente suspendido.
- **stop:** Termina la vida de un *thread*.
- **join:** Esperar hasta que un *thread* en particular haya terminado.

296

Otros temas de interés...

- Grupos de *threads*.
- Seguridad entre *threads*.
- Prioridad de los *threads*.
- *Threads* demonios.
- Depuración de *threads*.

297

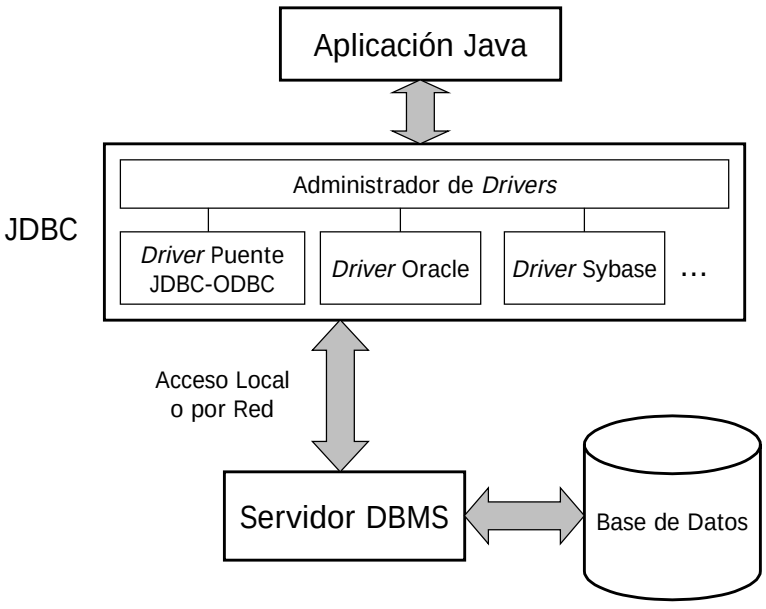
CAPÍTULO 9

Conectividad con Bases de Datos

JDBC

- **JDBC** es un API de Java para ejecutar enunciados **SQL** (*Structured Query Language*).
- Consiste un conjunto de clases e interfaces escritas en Java para tener acceso a datos contenidos en bases de datos relacionales.
- Es parte integral de Java desde la versión 1.1.

299



300

Clases de Interés

- **Driver**: Interface que al ser implementada provee los servicios necesarios para acceder a un sistema manejador de base de datos (DBMS) específico.
- **DriverManager**: Clase que provee los servicios básicos para administrar un conjunto de *drivers* JDBC.
- **Connection**: Interface que representa una sesión con una base de datos específica. Dentro del contexto de una conexión, se ejecutan enunciados SQL y se devuelven resultados.

301

- **Statement**: Un objeto que implementa esta interface se utiliza para ejecutar un enunciado estático de SQL y para obtener los resultados producidos.
- **ResultSet**: Interface que provee acceso a una tabla de datos generados al ejecutar un Statement. Los renglones de la tabla son recuperados en orden secuencial. Dentro de un mismo renglón, los valores de las columnas se pueden acceder en cualquier orden.

302

URL para Base de Datos

- Para establecer una conexión a una base de datos se requiere utilizar un URL con la siguiente forma:

  `jdbc:<subprotocolo>:<subnombre>`

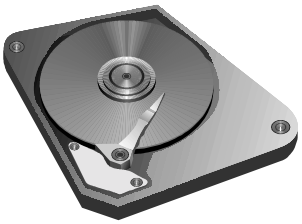
- El **`<subprotocolo>`** es el nombre del *driver* o del mecanismo de conexión.
- El **`<subnombre>`** es la forma de identificar a la base de datos y es dependiente del subprotocolo.
- Ejemplos:

  `jdbc:odbc:personal`

  `jdbc:msql://db.cem.itesm.mx:3000/depto`

303

Accediendo a una Base de Datos



1. Importar en nuestro programa el paquete **`java.sql`**.
2. Cargar la clase que implementa el *driver* JDBC del DBMS que se utilizará.
3. Establecer una conexión con la base de datos.
4. Crear un enunciado SQL.
5. Ejecutar una consulta o modificación sobre el enunciado.
6. En el caso haber realizado una consulta, procesar la tabla de resultados devuelta.
7. Cerrar el enunciado y la conexión.

304

Excepciones en JDBC

- La mayoría de las instrucciones que conforman el API de JDBC potencialmente pueden arrojar la excepción verificada **SQLException**, por lo que es importante lidiar adecuadamente con ella en el código (usando la instrucción **`try/catch`** o la cláusula **`throws`**).

305

Cargando el *driver* de JDBC.

- El método **Class.forName(String)** se utiliza para cargar dinámicamente la clase que implementa el driver JDBC de un DBMS particular:

```
Class.forName("ClaseDriverJDBC");
```

306

Establecer una Conexión

- Crear una nueva instancia de **Connection** utilizando el método de clase **`getConnection`** de la clase **DriverManager** mandando como argumento una cadena que contenga el URL correspondiente:

```
Connection conexion =
    DriverManager.getConnection(
        "URL-JDBC");
```

307

Crear un Enunciado SQL

- Sobre el objeto que representa nuestra conexión, aplicar el método **`createStatement`** para generar un nuevo **Statement** de SQL:

```
Statement enunciado =
    conexion.createStatement();
```

308

Crear una Tabla

- Utilizar el método **`executeUpdate`** sobre el objeto enunciado, mandando como argumento una cadena con el código SQL para crear una tabla:

```
enunciado.executeUpdate(
    "create table alumnos "
    + "(matricula int, "
    + "nombre char(30), "
    + "calif int)");
```

309

Eliminar una Tabla

- Utilizar el método **`executeUpdate`** sobre el objeto enunciado, mandando como argumento una cadena con el código SQL para eliminar una tabla:

```
enunciado.executeUpdate(
    "drop table alumnos");
```

310

Añadir un Renglón a una Tabla

- Utilizar el método **`executeUpdate`** sobre el objeto enunciado, mandando como argumento una cadena con el código SQL para insertar elementos a una tabla:

```
enunciado.executeUpdate(
    "insert into alumnos values "
    + "(441520, 'Juan', 85)");
```

311

Eliminar un Renglón de una Tabla

- Utilizar el método **`executeUpdate`** sobre el objeto enunciado, mandando como argumento una cadena con el código SQL para eliminar elementos de una tabla:

```
enunciado.executeUpdate(
   "delete from alumnos "
   + "where matricula = 441520");
```

312

Actualizar un Renglón de una Tabla

- Utilizar el método **`executeUpdate`** sobre el objeto enunciado, mandando como argumento una cadena con el código SQL para actualizar elementos de una tabla:

```
enunciado.executeUpdate(
   "update alumnos "
   + "set calif = 99 "
   + "where matricula = 441520");
```

313

Consultar una Tabla

- Utilizar el método **`executeQuery`** sobre el objeto enunciado, mandando como argumento una cadena con el código SQL para hacer una selección sobre una tabla.

314

- El resultado devuelto es un objeto de la clase **ResultSet**:

```
ResultSet resultado =
    enunciado.executeQuery(
        "select * from alumnos");
```

315

Procesar la Tabla Resultante

- Sobre el objeto resultado se pueden aplicar los siguientes métodos:
  - **`next()`** avanza al siguiente renglón de la tabla resultante. Devuelve **`false`** si ya no hay más renglones o **`true`** en caso contrario.
  - **`get`*`Tipo`*`(int n)`** devuelve como ***`Tipo`*** el elemento que se encuentra en la columna ***`n`*** (la primera columna es la 1) del renglón actual.

316

```
while (resultado.next()) {
    int matricula = resultado.getInt(1);
    String nombre = resultado.getString(2);
    int calif     = resultado.getInt(3);

    System.out.println(matricula
        + " " + nombre + " " + calif);
}
```

317

Cerrar el Enunciado y la Conexión

- Al finalizar se debe cerrar el enunciado SQL y la conexión:

```
enunciado.close();
conexion.close();
```

318

```
        int a = entrada.readInt();
        int b = entrada.readInt();
        int c = a + b;

        salida.writeInt(c);

        entrada.close();
        salida.close();
        soc.close();
        servidor.close();
    } // main
} // Servidor
```

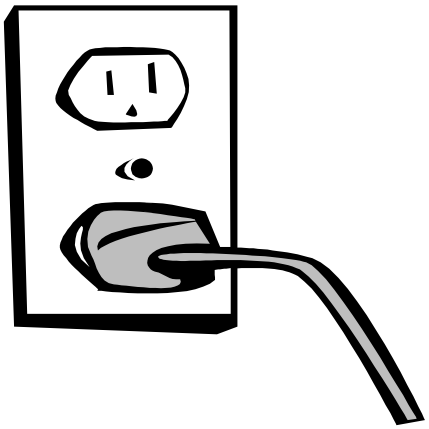
319

CAPÍTULO 10

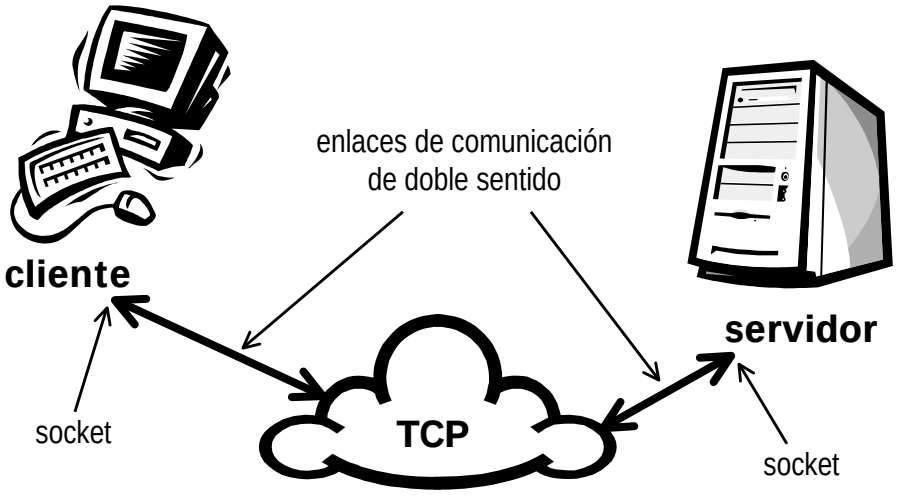
Sockets

Sockets

- Un *socket* es una abstracción de software que representa un extremo de un enlace de comunicación de doble sentido.



321



322

Puertos

- Las computadoras generalmente cuentan con una única conexión física a la red, por lo que se requieren **números de puerto** para poder distinguir qué datos son de qué aplicación.
- Los números de puerto van del 0 al 65535, sin embargo del 0 al 1023 están reservados para servicios predefinidos. Por ejemplo:

http	número de puerto 80
ftp	número de puerto 21
telnet	número de puerto 23

323

Flujos (*Streams*)

- Para transferir datos a través de los sockets se requiere utilizar flujos de entrada (lectura) y salida (escritura).
- Cada instancia de la clase **Socket** tiene asociado un flujo de entrada y uno de salida, los cuales se obtienen con los métodos **getInputStream()** y **getOutputStream()** respectivamente.

324

- Los tipos de flujos más útiles en la comunicación con socketes son:
  - **DataInputStream** y **DataOutputStream** para datos binarios que representan tipos primitivos (int, byte, boolean, Strings UTF, etc.).
  - **BufferedReader** y **PrintWriter** para información manipulada como líneas de texto.
  - **ObjectInputStream** y **ObjectOutputStream** para datos binarios que representan instancias de clases serializables.

325

Del Lado del Cliente

1. Crear una instancia de **Socket**, indicando el URL del *host* y número de puerto del servidor.
2. Abrir un flujo de entrada y/o salida asociándolo al socket.
3. Leer y/o escribir de los flujos según el protocolo del servidor.
4. Cerrar los flujos.
5. Cerrar el socket.

326

```
import java.io.*;
import java.net.*;

public class Cliente {
    public static void main(String[] args)
            throws IOException {
        Socket soc = new Socket(
            "servidor.cem.itesm.mx", 5555);
        DataOutputStream salida =
            new DataOutputStream(
                soc.getOutputStream());
        DataInputStream entrada =
            new DataInputStream(
                soc.getInputStream());
```

327

```
        salida.writeInt(10);
        salida.writeInt(5);

        int resultado = entrada.readInt();
        System.out.println(resultado);

        entrada.close();
        salida.close();
        soc.close();
    } // main
} // Cliente
```

328

Del Lado del Servidor

1. Crear una instancia de **ServerSocket**, indicando el el número de puerto.
2. Aceptar una comunicación con el cliente a través de un **Socket** convencional.
3. Abrir un flujo de entrada y/o salida asociándolo al socket.
4. Leer y/o escribir de los flujos.
5. Cerrar los flujos.
6. Cerrar el socket convencional y el ServerSocket.

329

```
import java.io.*;
import java.net.*;
public class Servidor {
    public static void main(String[] args)
            throws IOException {
        ServerSocket servidor =
            new ServerSocket(5555);
        Socket soc = servidor.accept();
        DataOutputStream salida =
            new DataOutputStream(
                soc.getOutputStream());
        DataInputStream entrada =
            new DataInputStream(
                soc.getInputStream());
```

330

```
        int a = entrada.readInt();
        int b = entrada.readInt();
        int c = a + b;

        salida.writeInt(c);

        entrada.close();
        salida.close();
        soc.close();
        servidor.close();
    } // main
} // Servidor
```

331

CAPÍTULO 11

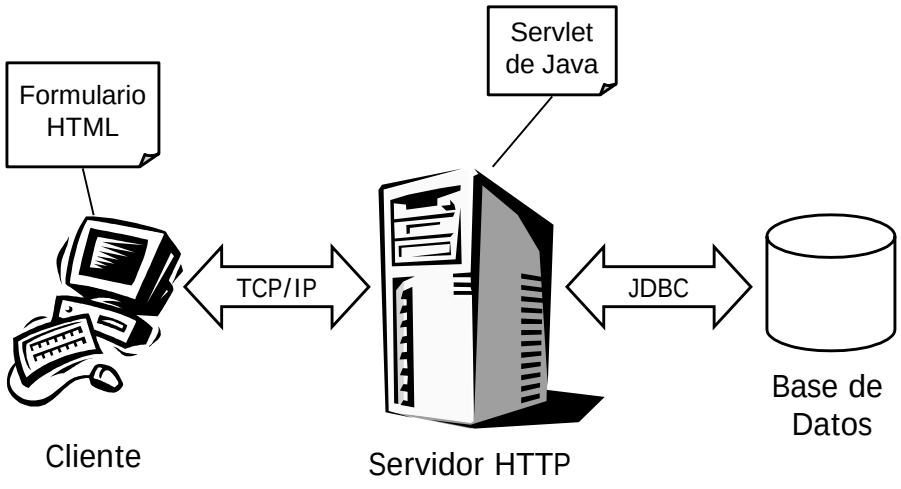
Servlets

Servlets

- Código que corre dentro de servidores y extienden su funcionalidad.
- Constituye un reemplazo eficiente de los scripts CGI.
- Requieren de servidores habilitados para Java que puedan reponder las solicitudes de los clientes.
- Los servelts son a los servidores lo que los applets son a los browsers.

333

Uso típico:



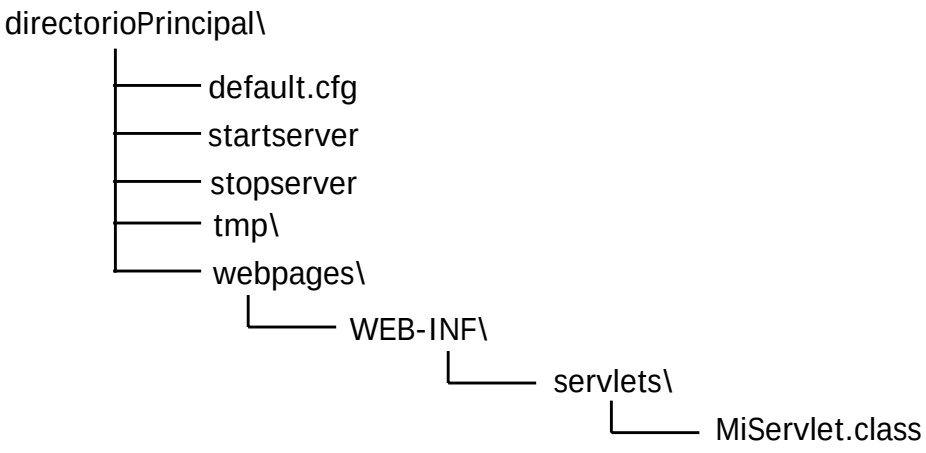
334

Ambiente para desarrollar Servlets (JSDK 2.1)

- Estructura de directorio.
- Archivo de configuración.
- Cliente (página de HTML).
- Servidor de Web que soporte Servlets.
- Servlet.

335

Estructura del Directorio



336

Archivo default.cfg

```
server.port=8080
server.hostname=
server.inet=
server.docbase=webpages
server.tempdir=tmp
```

337

Cliente

```
<FORM ACTION="servlet/MiServlet">
  Primer n&uacute;mero:
  <INPUT TYPE=TEXT NAME=num1>
  Segundo n&uacute;mero:
  <INPUT TYPE=TEXT NAME=num2>
  <P>
  <INPUT TYPE=SUBMIT VALUE="Continuar">
  <P>
  <INPUT TYPE=RESET VALUE="Borrar Datos">
</FORM>
```

338

Servidor

- Ejecutarlo desde “directorioPrincipal”.
- Usar el script “startserver” para echarlo a andar y “stopserver” para detenerlo.
- Cuando se desee probar un servlet modificado pero previamente cargado, es necesario detener el servidor y echarlo a andar otra vez.

339

Servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MiServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter salida = response.getWriter();
```

340

```
        int num1 = Integer.parseInt(
            request.getParameter("num1"));
        int num2 = Integer.parseInt(
            request.getParameter("num2"));

        salida.println("<html> <head><title>”
            + "Suma de dos números</title>"
            + "</body> Suma de: " + num1 + " + "
            + num2 + " = " + (num1 + num2)
            + "</body></html>");
        salida.close();
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

341

Rastreando una Sesión

- Distintos servlets pueden compartir información de un mismo usuario a través del tiempo.
- Para rastrear una sesión:
  - ✓ Obtener el objeto **HttpSession** del usuario.
  - ✓ Obtener y/o establecer datos usando el objeto **HttpSession**.
  - ✓ Invalidar la sesión (opcional).
- Para que esto funcione, el cliente debe aceptar **cookies**.

342

Oteniendo una Sesión

- Sobre el argumento de tipo **HttpServletRequest** del método **doGet** o **doPost**, aplicar el método **getSession**.

```
HttpSession session =
    request.getSession(true);
```

- El argumento **true** del método **getSessión** indica que se debe crear una sesión en caso de que no exista una previamente.
- El método **getSession** debe ser invocado antes que el método **getWriter**.

343

Estableciendo Datos en la Sesión

- Sobre el objeto de tipo **HttpSession** aplicar el método **putValue** indicando como argumentos el nombre (**String**) y el valor asociado (**Object**) a ese nombre.

```
session.putValue(
    "unNumero",
    new Integer(123));
```

344

Obteniendo Datos de la Sesión

- Sobre el objeto de tipo **HttpSession** aplicar el método **getValue** indicando como argumento un nombre (**String**).

```
Integer i = (Integer)
  session.getValue("unNumero");
```

- Se devuelve **null** si no hay un valor asociado al nombre.

345

Invalidando una Sesión

- Invalidar una sesión implica eliminar del sistema el objeto de tipo **HttpSession** junto con todos sus valores previamente establecidos.

```
session.invalidate();
```

- Las sesiones se invalidan automáticamente después de un cierto tiempo (30 minutos por omisión).

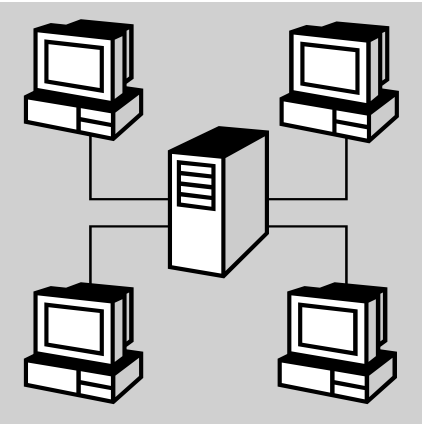
346

CAPÍTULO 12

RMI

Invocación Remota de Métodos

- RMI es el modelo de Java para desarrollar aplicaciones con objetos distribuidos.
- Una aplicación RMI se compone típicamente de dos programas separados: un cliente y un servidor.



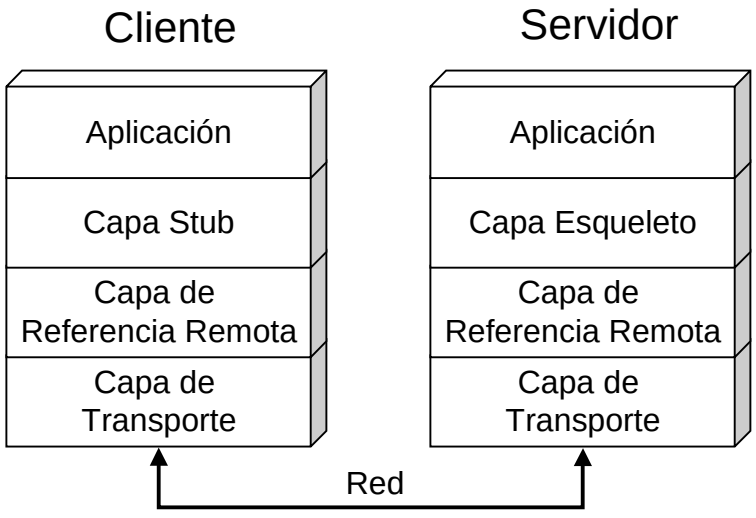
348

Invocación Remota de Métodos (cont.)

- Normalmente el sevidor crea uno o varios objetos remotos y los registra para hacerlos públicos.
- El cliente obtiene las referencias a los objetos remotos e invoca métodos sobre ellos de manera transparente como si fueran objetos locales.

349

Arquitectura RMI



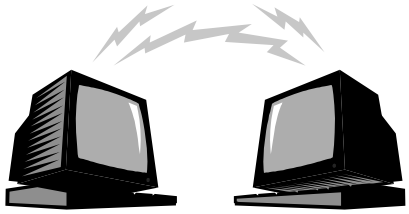
350

Arquitectura RMI (cont.)

- Las capas de **“stub”** y **“esqueleto”** se comportan como objetos sustitutos que esconden lo “remoto” de la invocación de un método.
- La capa de **referenca remota** se encarga de empaquetar la invocación del método, sus argumentos y valor de regreso para ser transportada por la red.
- La capa de **transporte** es la conexión de red entre los sistemas.

351

Usando RMI



1. Definir las funciones de la clase remota como una interface de Java.
2. Escribir la implementación del objeto remoto y del servidor.
3. Escribir el programa cliente que haga uso de los servicios remotos.

352

Definir la Interface Remota

- Debe ser pública
- Debe extender de la clase java.rmi.Remote
- Cada método debe declarar que arroja la excepción java.rmi.RemoteException

353

```
package fibo;

import java.rmi.*;

public interface InterfaceFibonacci
        extends Remote {
    int calculaFibonacci(int n)
        throws RemoteException;
}
```

354

Implementar un Objeto Remoto

- Debe extender de la clase java.rmi.server.UnicastRemoteObject
- Debe implementar la interface remota
- Debe definir un constructor que arroje la excepción java.rmi.RemoteException

355

```
package fibo;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class ObjetoFibonacci
        extends UnicastRemoteObject
        implements InterfaceFibonacci {

    private static Hashtable precalculados
        = new Hashtable();
```

356

```
    public ObjetoFibonacci()
            throws RemoteException {
        precalculados.put(new Integer(0),
            new Integer(1));
        precalculados.put(new Integer(1),
            new Integer(1));
    }

    public int calculaFibonacci(int n)
            throws RemoteException {
        Integer llave = new Integer(n);
        Integer valor =
            (Integer) precalculados.get(llave);
```

357

```
        if (valor != null) {
            System.out.println(
                "Ya calculado: fib(" + n
                + ") = " + valor);
            return valor.intValue();
        } else {
            int nuevo =
                calculaFibonacci(n - 1)
                + calculaFibonacci(n - 2);
            precalculados.put(llave,
                new Integer(nuevo));
            return nuevo;
        } // else
    } // calculaFibonacci
} // ObjetoFibonacci
```

358

El Servidor

- Debe contener el método main
- Debe instalar un administrador de seguridad
- Debe crear una instancia del objeto remoto
- Debe registrar el objeto remoto

359

```
package fibo;

import java.rmi.*;

public class Servidor {

    public static void main(String[] args)
            throws Exception {
        if (System.getSecurityManager()
               == null) {
            System.setSecurityManager(
                new RMISecurityManager());
        }
```

360

```
        String nombre =
            "//localhost:1099/FiboObj";
        ObjetoFibonacci obj =
            new ObjetoFibonacci();
        Naming.rebind(nombre, obj);
        System.out.println(
            "Objeto FiboObj ligado");
    } // main
} // Servidor
```

361

El Cliente

- Debe obtener una referencia al objeto remoto
- Utiliza la referencia al objeto remoto como si fuera un objeto local

362

```
package fibo;

import java.rmi.*;

public class Cliente {

    public static void main(String args[]) {

        try {
            String nombre =
                "//localhost:1099/FiboObj";
            InterfaceFibonacci intf =
                (InterfaceFibonacci)
                    Naming.lookup(nombre);
```

363

```
            int fib =
                intf.calculaFibonacci(20);
            System.out.println(fib);
          } catch (Exception e) {
            e.printStackTrace();
          } // try
      } // main
  } // Cliente
```

364

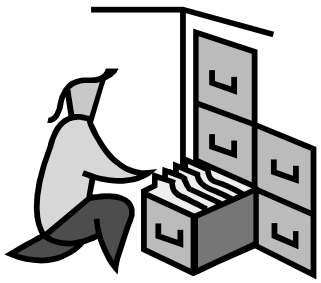
Compilando todo...

- Desde el directorio “d:\dir\” (el directorio que contiene al directorio “fibo” que contiene los archivos fuente):

```
javac fibo\InterfaceFibonacci.java
javac fibo\ObjetoFibonacci.java
javac fibo\Servidor.java
javac fibo\Cliente.java
rmic -d . fibo.ObjetoFibonacci
```

365

El Registro



- El registro (*registry*) es un servidor de nombres que permite a los clientes obtener referencias a los objetos remotos.
- Por omisión escucha desde el puerto 1099.
- Para ejecutarlo (desde “d:\dir\”):

```
rmiregistry
```

366

Ejecutando el Servidor

- Crear un archivo de políticas de acceso llamado “policy” en “d:\dir\”. Por ejemplo:

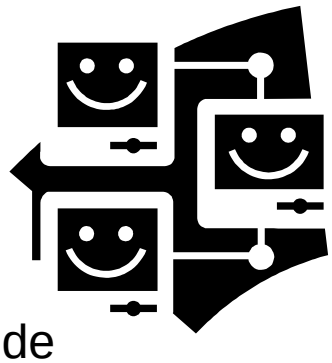
```
grant {
  permission java.security.AllPermission;
};
```

- Ejecutar el servidor desde “d:\dir\”:

```
java -Djava.security.policy=d:\dir\policy
  fibo.Servidor
```

367

Ejecutando el Cliente



- Ejecutar el client desde “d:\dir\”:

```
java -Djava.security.policy=d:\dir\policy
  fibo.Cliente
```

368